Contents

About this book xi
Who should use this book ix
How this book is structured ix
How to read the syntax diagrams xi
Conventions used in this manual xiii
Error conditions xiii
Highlighting conventions xiii
Conventions describing Unicode data xiii
Related documentation xiii

Chapter 1. Concepts 1
Databases 1
Structured Query Language (SQL) 1
Queries and table expressions 2
Introduction to DB2 Call Level Interface and ODBC 2
Java application development for IBM data servers 4
Schemas 5
Tables 6
Types of tables 6
Constraints 8
Indexes 8
Triggers 10
Views 11
Aliases 13
Package 13
Authorization, privileges, and object ownership 13
System catalog views 19
Application processes, concurrency, and recovery 19
Isolation levels 21
Table spaces 26
Character conversion 28
Multicultural support and SQL statements 30
Connecting to distributed relational databases 31
Event monitors that write to tables, files, and pipes 32
Database partitioning across multiple database partitions 33
Large object behavior in partitioned tables 34
DB2 federated systems 35
Federated systems 35
What is a data source? 36
The federated database 37
The SQL compiler 37
Wrappers and wrapper modules 37
Server definitions and server options 38
User mappings 39
Nicknames and data source objects 40
Nickname column options 40
Data type mappings 41
The federated server 42
Supported data sources 43
The federated database system catalog 45
The query optimizer 46
Collating sequences 48

Chapter 2. Language elements 51
Characters 52
Tokens 53
Identifiers 54
Data types 80
Data type list 81
Promotion of data types 104
Casting between data types 106
Assignments and comparisons 114
Rules for result data types 130
Rules for string conversions 135
String comparisons in a Unicode database 136
Resolving the anchor object for an anchored type 138
Resolving the anchor object for an anchored row type 140
Database partition-compatible data types 142
Constants 144
Special registers 149
CURRENT CLIENT_ACCTNG 153
CURRENT CLIENT_APPLNAME 154
CURRENT CLIENT_USERID 155
CURRENT CLIENT_WRKSTNNAME 156
CURRENT DATE 157
CURRENT DBPARTITIONNUM 158
CURRENT DECIMALROUNDINGMODE 159
CURRENT DEFAULTTRANSFORMGROUP 160
CURRENT DEGREE 161
CURRENT EXPLAIN MODE 162
CURRENT EXPLAIN SNAPSHOT 163
CURRENT FEDERATEDASYNCRONY 164
CURRENT IMPLICITXMLPARSEOPTION 165
CURRENT ISOLATION 166
CURRENT LOCALELC_MESSAGES 167
CURRENT LOCALELC_TIME 168
CURRENT LOCKTIMEOUT 169
CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION 170
CURRENT MDCROLLOUTMODE 171
CURRENT OPTIMIZATION PROFILE 172
CURRENT PACKAGEPATH 173
CURRENT PATH 174
CURRENT QUERYOPTIMIZATION 175
CURRENT REFRESHAGE 176
CURRENT SCHEMA 177
CURRENT SERVER 178
CURRENT SQLCCFLAGS 179
CURRENT TIME 180
CURRENT TIMESTAMP 181
CURRENT TIMEZONE 182
CURRENT USER 183
SESSION_USER 184
SYSTEM_USER 185
USER 186
Global variables 187
Functions 190
Methods 204
Conservative binding semantics ............................................. 212
Expressions ................................................................. 215
Datetime operations and durations .............................. 226
CASE expression ......................................................... 231
CAST specification ..................................................... 234
Field reference ............................................................. 239
XMLCAST specification ................................................ 240
ARRAY element specification ........................................ 242
Array constructor ......................................................... 243
Dereference operation ................................................... 245
Method invocation ......................................................... 247
OLAP specifications ....................................................... 265
ROW CHANGE expression ............................................ 258
Sequence reference ....................................................... 260
Subtype treatment .......................................................... 264
Determining data types of untyped expressions ............ 265
Row expression ............................................................... 272
Predicates ................................................................. 273
Predicate processing for queries .................................... 274
Search conditions .......................................................... 277
Basic predicate ............................................................. 280
Quantified predicate ...................................................... 281
ARRAY_EXISTS ............................................................ 284
BETWEEN predicate ....................................................... 285
Cursor predicates ........................................................ 286
EXISTS predicate .......................................................... 288
IN predicate ................................................................. 289
LIKE predicate .............................................................. 291
NULL predicate ............................................................. 296
TYPE predicate ............................................................ 297
VALIDATED predicate ................................................... 298
XMLEXIST predicate ..................................................... 300

Chapter 3. Functions .................................................. 303
Functions overview ....................................................... 303
Supported functions and administrative SQL .......... 304
routines and views ......................................................... 304
Aggregate functions .................................................... 314
ARRAY_AGG ............................................................... 316
AVG ................................................................. 318
CORRELATION ............................................................ 320
COUNT ................................................................. 321
COUNT_BIG ............................................................... 322
COVARIANCE ............................................................ 324
GROUPING ............................................................... 325
MAX ................................................................. 327
MIN ................................................................. 328
Regression functions ................................................... 329
STDDEV ............................................................... 332
SUM ................................................................. 333
VARIANCE ............................................................... 334
XMLAGG ............................................................... 335
XMLGROUP ............................................................. 337
Scalar functions .......................................................... 339
ABS or ABSVAL .......................................................... 341
ACOS ................................................................. 342
ADD_MONTHS ............................................................. 343
ARRAY_DELETE .......................................................... 345
ARRAY_FIRST ............................................................ 346
ARRAY_LAST ............................................................. 347
ARRAY_NEXT ............................................................ 348
ARRAY_PRIOR ............................................................ 349
ASCII ................................................................. 350
ASIN ................................................................. 351
ATAN ................................................................. 352
ATAN2 ................................................................. 353
ATANH ................................................................. 354
BITINT ................................................................. 355
BITAND, BITANDNOT, BITOR, BITXOR, and ...... 357
BITNOT ................................................................. 357
BLOB ................................................................. 359
CARDINALITY ............................................................ 360
CEILING or CEIL ........................................................ 361
CHAR ................................................................. 362
CHARACTER_LENGTH .................................................. 368
CHR ................................................................. 370
CLOB ................................................................. 371
COALESCE .............................................................. 372
COLLATION_KEY_BIT .................................................. 373
COMPARE_DECFLT ...................................................... 375
CONCAT ............................................................... 377
COS ................................................................. 378
COSH ................................................................. 379
COT ................................................................. 380
CURSOR_ROWCOUNT .................................................... 381
DATAPARTITIONNUM .................................................... 382
DATE ................................................................. 383
DAY ................................................................. 384
DAYNAME ............................................................... 385
DAYOFWEEK ............................................................. 386
DAYOFWEEK_ISO ........................................................ 387
DAYOFYEAR ............................................................. 388
DAYS ................................................................. 389
DBCLOB ............................................................... 390
DBPARTITIONNUM ....................................................... 391
DECFLT ............................................................... 393
DECFLT_FORMAT ........................................................ 395
DECIMAL or DEC ........................................................ 397
DECODE ............................................................... 401
DECRYPT_BIN and DECRYPT_CHAR ....................... 403
DEGREES ............................................................... 405
DEREF ............................................................... 406
DIFFERENCE ............................................................ 407
DIGITS ............................................................... 408
DOUBLE_PRECISION or DOUBLE .......... 409
EMPTY_BLOB, EMPTY_CLOB, and .. 411
EMPTY_DBCLOB ........................................................ 411
ENCRYPT ............................................................... 412
EVENT_MON_STATE ..................................................... 415
EXP ................................................................. 416
EXTRACT ............................................................... 417
FLOAT ............................................................... 419
FLOOR ............................................................... 420
GENERATE_UNIQUE ...................................................... 421
GETHINT ............................................................... 423
GRAPHIC ............................................................. 424
GREATEST .............................................................. 429
HASEDVALUE ............................................................ 430
HEX ................................................................. 432
HOUR ............................................................... 434
IDENTITY_VAL_LOCAL ............................................... 435
INITCAP ............................................................... 439
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>INSERT</td>
<td>441</td>
</tr>
<tr>
<td>INSTR</td>
<td>445</td>
</tr>
<tr>
<td>INTEGER or INT</td>
<td>446</td>
</tr>
<tr>
<td>JULIAN_DAY</td>
<td>448</td>
</tr>
<tr>
<td>LAST_DAY</td>
<td>449</td>
</tr>
<tr>
<td>LCASE</td>
<td>450</td>
</tr>
<tr>
<td>LCASE (locale sensitive)</td>
<td>451</td>
</tr>
<tr>
<td>LEAST</td>
<td>452</td>
</tr>
<tr>
<td>LEFT</td>
<td>453</td>
</tr>
<tr>
<td>LENGTH</td>
<td>456</td>
</tr>
<tr>
<td>LN</td>
<td>458</td>
</tr>
<tr>
<td>LOCATE</td>
<td>459</td>
</tr>
<tr>
<td>LOCATE_IN_STRING</td>
<td>463</td>
</tr>
<tr>
<td>LOG10</td>
<td>466</td>
</tr>
<tr>
<td>LONG_VARCHAR</td>
<td>467</td>
</tr>
<tr>
<td>LONG_VARGRAPHIC</td>
<td>468</td>
</tr>
<tr>
<td>LOWER</td>
<td>469</td>
</tr>
<tr>
<td>LOWER (locale sensitive)</td>
<td>470</td>
</tr>
<tr>
<td>LPAD</td>
<td>472</td>
</tr>
<tr>
<td>LTRIM</td>
<td>475</td>
</tr>
<tr>
<td>MAX</td>
<td>476</td>
</tr>
<tr>
<td>MAX_CARDINALITY</td>
<td>477</td>
</tr>
<tr>
<td>MICROSECOND</td>
<td>478</td>
</tr>
<tr>
<td>MIDNIGHT_SECONDS</td>
<td>479</td>
</tr>
<tr>
<td>MIN</td>
<td>480</td>
</tr>
<tr>
<td>MINUTE</td>
<td>481</td>
</tr>
<tr>
<td>MOD</td>
<td>482</td>
</tr>
<tr>
<td>MONTH</td>
<td>483</td>
</tr>
<tr>
<td>MONTHNAME</td>
<td>484</td>
</tr>
<tr>
<td>MONTHS_BETWEEN</td>
<td>485</td>
</tr>
<tr>
<td>MULTIPLY_ALT</td>
<td>487</td>
</tr>
<tr>
<td>NEXT_DAY</td>
<td>489</td>
</tr>
<tr>
<td>NORMALIZE_DECFLOAT</td>
<td>491</td>
</tr>
<tr>
<td>NULLIF</td>
<td>492</td>
</tr>
<tr>
<td>NVL</td>
<td>493</td>
</tr>
<tr>
<td>OCTET_LENGTH</td>
<td>494</td>
</tr>
<tr>
<td>OVERLAY</td>
<td>495</td>
</tr>
<tr>
<td>PARAMETER</td>
<td>499</td>
</tr>
<tr>
<td>POSITION</td>
<td>500</td>
</tr>
<tr>
<td>POSSTR</td>
<td>503</td>
</tr>
<tr>
<td>POWER</td>
<td>505</td>
</tr>
<tr>
<td>QUANTIZE</td>
<td>506</td>
</tr>
<tr>
<td>QUARTER</td>
<td>508</td>
</tr>
<tr>
<td>RADIANS</td>
<td>509</td>
</tr>
<tr>
<td>RAISE_ERROR</td>
<td>510</td>
</tr>
<tr>
<td>RAND</td>
<td>511</td>
</tr>
<tr>
<td>REAL</td>
<td>514</td>
</tr>
<tr>
<td>REC2XML</td>
<td>518</td>
</tr>
<tr>
<td>REPEAT</td>
<td>519</td>
</tr>
<tr>
<td>REPLACE</td>
<td>521</td>
</tr>
<tr>
<td>RID_BIT and RID</td>
<td>523</td>
</tr>
<tr>
<td>RIGHT</td>
<td>526</td>
</tr>
<tr>
<td>ROUND</td>
<td>532</td>
</tr>
<tr>
<td>ROUND_TIMESTAMP</td>
<td>534</td>
</tr>
<tr>
<td>RPAD</td>
<td>537</td>
</tr>
<tr>
<td>SECLABEL</td>
<td>538</td>
</tr>
<tr>
<td>SECLABEL_BY_NAME</td>
<td>539</td>
</tr>
<tr>
<td>SECLABEL_TO_CHAR</td>
<td>540</td>
</tr>
<tr>
<td>SECOND</td>
<td>542</td>
</tr>
<tr>
<td>SIGN</td>
<td>544</td>
</tr>
<tr>
<td>SIN</td>
<td>545</td>
</tr>
<tr>
<td>SINH</td>
<td>546</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>547</td>
</tr>
<tr>
<td>SOUNDDEX</td>
<td>548</td>
</tr>
<tr>
<td>SPACE</td>
<td>549</td>
</tr>
<tr>
<td>SQRT</td>
<td>550</td>
</tr>
<tr>
<td>STRIP</td>
<td>551</td>
</tr>
<tr>
<td>SUBSTR</td>
<td>552</td>
</tr>
<tr>
<td>SUBSTRB</td>
<td>555</td>
</tr>
<tr>
<td>SUBSTRING</td>
<td>558</td>
</tr>
<tr>
<td>TABLE_NAME</td>
<td>560</td>
</tr>
<tr>
<td>TABLE_SCHEMA</td>
<td>561</td>
</tr>
<tr>
<td>TAN</td>
<td>563</td>
</tr>
<tr>
<td>TANH</td>
<td>564</td>
</tr>
<tr>
<td>TIME</td>
<td>565</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>566</td>
</tr>
<tr>
<td>TIMESTAMP_FORMAT</td>
<td>568</td>
</tr>
<tr>
<td>TIMESTAMP_ISO</td>
<td>575</td>
</tr>
<tr>
<td>TIMESTAMPDIFF</td>
<td>576</td>
</tr>
<tr>
<td>TO_CHAR</td>
<td>578</td>
</tr>
<tr>
<td>TO_CLOB</td>
<td>579</td>
</tr>
<tr>
<td>TO_DATE</td>
<td>580</td>
</tr>
<tr>
<td>TO_NUMBER</td>
<td>581</td>
</tr>
<tr>
<td>TO_TIMESTAMP</td>
<td>582</td>
</tr>
<tr>
<td>TOTALORDER</td>
<td>583</td>
</tr>
<tr>
<td>TRANSLATE</td>
<td>585</td>
</tr>
<tr>
<td>TRIM</td>
<td>588</td>
</tr>
<tr>
<td>TRIM_ARRAY</td>
<td>590</td>
</tr>
<tr>
<td>TRUNCATE or TRUNC</td>
<td>593</td>
</tr>
<tr>
<td>TYPE_ID</td>
<td>596</td>
</tr>
<tr>
<td>TYPE_NAME</td>
<td>597</td>
</tr>
<tr>
<td>XMLATTRIBUTES</td>
<td>600</td>
</tr>
<tr>
<td>UPPER</td>
<td>601</td>
</tr>
<tr>
<td>UPPER (locale sensitive).</td>
<td>602</td>
</tr>
<tr>
<td>VALUE</td>
<td>604</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>605</td>
</tr>
<tr>
<td>VARCHAR_BIT_FORMAT</td>
<td>610</td>
</tr>
<tr>
<td>VARCHAR_FORMAT</td>
<td>611</td>
</tr>
<tr>
<td>VARCHAR_FORMAT_BIT</td>
<td>619</td>
</tr>
<tr>
<td>VARGRAPHIC</td>
<td>620</td>
</tr>
<tr>
<td>WEEK</td>
<td>626</td>
</tr>
<tr>
<td>WEEK_ISO</td>
<td>627</td>
</tr>
<tr>
<td>XML_ATTRIBUTES</td>
<td>628</td>
</tr>
<tr>
<td>XMLCOMMENT</td>
<td>630</td>
</tr>
<tr>
<td>XMLCONCAT</td>
<td>631</td>
</tr>
<tr>
<td>XMLDOCUMENT</td>
<td>632</td>
</tr>
<tr>
<td>XMLELEMENT</td>
<td>634</td>
</tr>
<tr>
<td>XMLFOREST</td>
<td>641</td>
</tr>
<tr>
<td>XMLNAMESPACES</td>
<td>644</td>
</tr>
<tr>
<td>XMLPARSE</td>
<td>646</td>
</tr>
<tr>
<td>XMLPI</td>
<td>649</td>
</tr>
<tr>
<td>XMLQUERY</td>
<td>650</td>
</tr>
<tr>
<td>XMLROW</td>
<td>653</td>
</tr>
<tr>
<td>XMLSERIALIZE</td>
<td>655</td>
</tr>
<tr>
<td>XMLEXPECT</td>
<td>657</td>
</tr>
<tr>
<td>XMLVALIDATE</td>
<td>659</td>
</tr>
<tr>
<td>XMLXSROBJECTID</td>
<td>663</td>
</tr>
<tr>
<td>XSLTRANSFORM</td>
<td>664</td>
</tr>
</tbody>
</table>
## Appendix E. Federated systems

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Valid server types in SQL statements</td>
<td>976</td>
</tr>
<tr>
<td>Function mapping options for federated systems</td>
<td>977</td>
</tr>
<tr>
<td>Default forward data type mappings</td>
<td>978</td>
</tr>
<tr>
<td>Default forward data type mappings for DB2 Database for Linux, UNIX, and Windows data sources</td>
<td>979</td>
</tr>
<tr>
<td>Default forward data type mappings for System i data sources</td>
<td>980</td>
</tr>
<tr>
<td>Default forward data type mappings for VM and VSE data sources</td>
<td>981</td>
</tr>
</tbody>
</table>

Default forward data type mappings for z/OS data sources | 982  
Default forward data type mappings for Informix data sources | 983  
Default forward data type mappings for Microsoft SQL Server data sources | 985  
Default forward data type mappings for ODBC data sources | 987  
Default forward data type mappings for Oracle NET8 data sources | 988  
Default forward data type mappings for Sybase data sources | 989  
Default forward data type mappings for Teradata data sources | 991  

Default reverse data type mappings | 992  
Default reverse data type mappings for DB2 Database for Linux, UNIX, and Windows data sources | 993  
Default reverse data type mappings for System i data sources | 994  
Default reverse data type mappings for VM and VSE data sources | 995  
Default reverse data type mappings for z/OS data sources | 996  
Default reverse data type mappings for Informix data sources | 997  
Default reverse data type mappings for Microsoft SQL Server data sources | 998  
Default reverse data type mappings for Oracle NET8 data sources | 999  
Default reverse data type mappings for Sybase data sources | 1000 
Default reverse data type mappings for Teradata data sources | 1001 

## Appendix F. The SAMPLE database

## Appendix G. Reserved schema names and reserved words

## Appendix H. Examples of interaction between triggers and referential constraints

## Appendix I. Explain tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADVISE_INDEX table</td>
<td>1038</td>
</tr>
<tr>
<td>ADVISE_INSTANCE table</td>
<td>1042</td>
</tr>
<tr>
<td>ADVISE_MQT table</td>
<td>1043</td>
</tr>
<tr>
<td>ADVISE_PARTITION table</td>
<td>1045</td>
</tr>
<tr>
<td>ADVISE_TABLE table</td>
<td>1047</td>
</tr>
<tr>
<td>ADVISE_WORKLOAD table</td>
<td>1048</td>
</tr>
<tr>
<td>EXPLAIN_ACTUALS table</td>
<td>1049</td>
</tr>
<tr>
<td>EXPLAIN_ARGUMENT table</td>
<td>1050</td>
</tr>
<tr>
<td>EXPLAIN_DIAGNOSTIC table</td>
<td>1058</td>
</tr>
<tr>
<td>EXPLAIN_DIAGNOSTIC_DATA table</td>
<td>1059</td>
</tr>
<tr>
<td>EXPLAIN_INSTANCE table</td>
<td>1060</td>
</tr>
<tr>
<td>EXPLAIN_OBJECT table</td>
<td>1063</td>
</tr>
<tr>
<td>EXPLAIN_OPERATOR table</td>
<td>1066</td>
</tr>
<tr>
<td>EXPLAIN_PREDICATE table</td>
<td>1068</td>
</tr>
</tbody>
</table>
About this book

The SQL Reference in its two volumes defines the SQL language used by DB2® Database for Linux®, UNIX®, and Windows®. It includes:

- Information about relational database concepts, language elements, functions, and the forms of queries (Volume 1)
- Information about the syntax and semantics of SQL statements (Volume 2)

Who should use this book

This book is intended for anyone who wants to use the Structured Query Language (SQL) to access a database. It is primarily for programmers and database administrators, but it can also be used by those who access databases through the command line processor (CLP).

This book is a reference rather than a tutorial. It assumes that you will be writing application programs and therefore presents the full functions of the database manager.

How this book is structured

The first volume of the SQL Reference contains information about relational database concepts, language elements, functions, and the forms of queries. The specific chapters and appendixes in that volume are briefly described here.

- “Concepts” discusses the basic concepts of relational databases and SQL.
- “Language elements” describes the basic syntax of SQL and the language elements that are common to many SQL statements.
- “Functions” contains syntax diagrams, semantic descriptions, rules, and usage examples of SQL aggregate and scalar functions.
- “Procedures” contains syntax diagrams, semantic descriptions, rules, and usage examples of procedures.
- “SQL queries” describes the various forms of a query.
- “SQL and XML limits” lists the SQL limitations.
- “SQLCA (SQL communications area)” describes the SQLCA structure.
- “SQLDA (SQL descriptor area)” describes the SQLDA structure.
- “System catalog views” describes the system catalog views.
- “Federated systems” describes options and type mappings for federated systems.
- “The SAMPLE database” introduces the SAMPLE database, which contains the tables that are used in many examples.
- “Reserved schema names and reserved words” contains the reserved schema names and the reserved words for the IBM® SQL and ISO/ANSI SQL2003 standards.
- “Examples of interaction between triggers and referential constraints” discusses the interaction of triggers and referential constraints.
- “Explain tables” describes the explain tables.
- “Explain register values” describes the interaction of the CURRENT EXPLAIN MODE and CURRENT EXPLAIN SNAPSHOT special register values with each other and with the PREP and BIND commands.
How this book is structured

- “Exception tables” contains information about user-created tables that are used with the SET INTEGRITY statement.
- “SQL statements allowed in routines” lists the SQL statements that are allowed to execute in routines with different SQL data access contexts.
- “CALL invoked from a compiled statement” describes the CALL statement that can be invoked from a compiled statement.
How to read the syntax diagrams

SQL syntax is described using the structure defined as follows:

Read the syntax diagrams from left to right and top to bottom, following the path of the line.

The ▶ symbol indicates the beginning of a syntax diagram.

The ── symbol indicates that the syntax is continued on the next line.

The ▶ symbol indicates that the syntax is continued from the previous line.

The ── symbol indicates the end of a syntax diagram.

Syntax fragments start with the ┌ symbol and end with the ┘ symbol.

Required items appear on the horizontal line (the main path).

Required items appear below the main path.

If an optional item appears above the main path, that item has no effect on execution, and is used only for readability.

If you can choose from two or more items, they appear in a stack.

If you must choose one of the items, one item of the stack appears on the main path.

If choosing one of the items is optional, the entire stack appears below the main path.

If one of the items is the default, it will appear above the main path, and the remaining choices will be shown below.
How to read the syntax diagrams

An arrow returning to the left, above the main line, indicates an item that can be repeated. In this case, repeated items must be separated by one or more blanks.

If the repeat arrow contains a comma, you must separate repeated items with a comma.

A repeat arrow above a stack indicates that you can make more than one choice from the stacked items or repeat a single choice.

Keywords appear in uppercase (for example, FROM). They must be spelled exactly as shown. Variables appear in lowercase (for example, column-name). They represent user-supplied names or values in the syntax.

If punctuation marks, parentheses, arithmetic operators, or other such symbols are shown, you must enter them as part of the syntax.

Sometimes a single variable represents a larger fragment of the syntax. For example, in the following diagram, the variable parameter-block represents the whole syntax fragment that is labeled parameter-block:

Adjacent segments occurring between “large bullets” (●) may be specified in any sequence.

The above diagram shows that item2 and item3 may be specified in either order. Both of the following are valid:

- required_item item1 item2 item3 item4
- required_item item1 item3 item2 item4
Conventions used in this manual

Error conditions
An error condition is indicated within the text of the manual by listing the SQLSTATE associated with the error in parentheses. For example:
A duplicate signature returns an SQL error (SQLSTATE 42723).

Highlighting conventions
The following conventions are used in this book.

| **Bold** | Indicates commands, keywords, and other items whose names are predefined by the system. |
| **Italics** | Indicates one of the following: |
| | • Names or values (variables) that must be supplied by the user |
| | • General emphasis |
| | • The introduction of a new term |
| | • A reference to another source of information |

Conventions describing Unicode data
When a specific Unicode code point is referenced, it is expressed as U+n where n is four to six hexadecimal digits, using the digits 0-9 and uppercase letters A-F. Leading zeros are omitted unless the code point would have fewer than four hexadecimal digits. The space character, for example, is expressed as U+0020. In most cases, the n value is the same as the UTF-16BE encoding.

Related documentation
The following publications might prove useful when you are preparing applications:

- **Getting Started with Database Application Development**
  - Provides an introduction to DB2 application development, including platform prerequisites; supported development software; and guidance on the benefits and limitations of the supported programming APIs.

- **DB2 for i5/OS SQL Reference**
  - This book defines SQL as supported by DB2 Query Manager and SQL Development Kit on System i®. It contains reference information for the tasks of system administration, database administration, application programming, and operation. This manual includes syntax, usage notes, keywords, and examples for each of the SQL statements used on i5/OS® systems running DB2.

- **DB2 for z/OS SQL Reference**
  - This book defines SQL used in DB2 for z/OS®. It provides query forms, SQL statements, SQL procedure statements, DB2 limits, SQLCA, SQLDA, catalog tables, and SQL reserved words for z/OS systems running DB2.

- **DB2 Spatial Extender User’s Guide and Reference**
  - This book discusses how to write applications to create and use a geographic information system (GIS). Creating and using a GIS involves supplying a database with resources and then querying the data to obtain information such as locations, distances, and distributions within areas.

- **IBM SQL Reference**
Related documentation

- This book contains all the common elements of SQL that span IBM's database products. It provides limits and rules that assist in preparing portable programs using IBM databases. This manual provides a list of SQL extensions and incompatibilities among the following standards and products: SQL92E, XPG4-SQL, IBM-SQL, and the IBM relational database products.

- **American National Standard X3.135-1992, Database Language SQL**
  - Contains the ANSI standard definition of SQL.

- **ISO/IEC 9075:1992, Database Language SQL**
  - Contains the 1992 ISO standard definition of SQL.

  - Contains a large portion of the 2003 ISO standard definition of SQL.

  - Contains the 2003 ISO standard definition for SQL procedure control statements.
Chapter 1. Concepts

Databases

A DB2 database is a relational database. The database stores all data in tables that are related to one another. Relationships are established between tables such that data is shared and duplication is minimized.

A relational database is a database that is treated as a set of tables and manipulated in accordance with the relational model of data. It contains a set of objects used to store, manage, and access data. Examples of such objects are tables, views, indexes, functions, triggers, and packages. Objects can be either defined by the system (system-defined objects) or defined by the user (user-defined objects).

A distributed relational database consists of a set of tables and other objects that are spread across different but interconnected computer systems. Each computer system has a relational database manager to manage the tables in its environment. The database managers communicate and cooperate with each other in a way that allows a given database manager to execute SQL statements on another computer system.

A partitioned relational database is a relational database whose data is managed across multiple database partitions. This separation of data across database partitions is transparent to most SQL statements. However, some data definition language (DDL) statements take database partition information into consideration (for example, CREATE DATABASE PARTITION GROUP). DDL is the subset of SQL statements used to describe data relationships in a database.

A federated database is a relational database whose data is stored in multiple data sources (such as separate relational databases). The data appears as if it were all in a single large database and can be accessed through traditional SQL queries. Changes to the data can be explicitly directed to the appropriate data source.

Structured Query Language (SQL)

SQL is a standardized language for defining and manipulating data in a relational database. In accordance with the relational model of data, the database is treated as a set of tables, relationships are represented by values in tables, and data is retrieved by specifying a result table that can be derived from one or more base tables.

SQL statements are executed by a database manager. One of the functions of the database manager is to transform the specification of a result table into a sequence of internal operations that optimize data retrieval. The transformation occurs in two phases: preparation and binding.

All executable SQL statements must be prepared before they can be executed. The result of preparation is the executable or operational form of the statement. The method of preparing an SQL statement and the persistence of its operational form distinguish static SQL from dynamic SQL.
Queries and table expressions

Queries and table expressions

A *query* is a component of certain SQL statements; it specifies a (temporary) result table.

A *table expression* creates a temporary result table from a simple query. Clauses further refine the result table. For example, you can use a table expression as a query to select all of the managers from several departments, specify that they must have over 15 years of working experience, and be located at the New York branch office.

A *common table expression* is like a temporary view within a complex query. It can be referenced in other places within the query, and can be used in place of a view. Each use of a specific common table expression within a complex query shares the same temporary view.

Recursive use of a common table expression within a query can be used to support applications such as airline reservation systems, bill of materials (BOM) generators, and network planning.

Introduction to DB2 Call Level Interface and ODBC

DB2 Call Level Interface (DB2 CLI) is IBM’s callable SQL interface to the DB2 family of database servers. It is a ‘C’ and ‘C++’ application programming interface for relational database access that uses function calls to pass dynamic SQL statements as function arguments.

You can use the DB2 CLI interface to access the following IBM data server databases:

- DB2 Version 9 for Linux, UNIX, and Windows
- DB2 Universal Database™ (DB2 UDB) Version 8 (and later) for Linux, UNIX, and Windows
- DB2 Universal Database Version 8 (and later) for OS/390® and z/OS
- DB2 for IBM i 5.4 and later

DB2 CLI is an alternative to embedded dynamic SQL, but unlike embedded SQL, it does not require host variables or a precompiler. Applications can be run against a variety of databases without having to be compiled against each of these databases. Applications use procedure calls at run time to connect to databases, issue SQL statements, and retrieve data and status information.

The DB2 CLI interface provides many features not available in embedded SQL. For example:

- CLI provides function calls that support a way of querying database catalogs that is consistent across the DB2 family. This reduces the need to write catalog queries that must be tailored to specific database servers.
- CLI provides the ability to scroll through a cursor:
  - Forward by one or more rows
  - Backward by one or more rows
  - Forward from the first row by one or more rows
  - Backward from the last row by one or more rows
  - From a previously stored location in the cursor.
Introduction to DB2 Call Level Interface and ODBC

- Stored procedures called from application programs that were written using CLI can return result sets to those programs.

DB2 CLI is based on the Microsoft® Open Database Connectivity (ODBC) specification, and the International Standard for SQL/CLI. These specifications were chosen as the basis for the DB2 Call Level Interface in an effort to follow industry standards and to provide a shorter learning curve for those application programmers already familiar with either of these database interfaces. In addition, some DB2 specific extensions have been added to help the application programmer specifically exploit DB2 features.

The DB2 CLI driver also acts as an ODBC driver when loaded by an ODBC driver manager. It conforms to ODBC 3.51.

DB2 CLI Background information

To understand DB2 CLI or any callable SQL interface, it is helpful to understand what it is based on, and to compare it with existing interfaces.

The X/Open Company and the SQL Access Group jointly developed a specification for a callable SQL interface referred to as the X/Open Call Level Interface. The goal of this interface is to increase the portability of applications by enabling them to become independent of any one database vendor’s programming interface. Most of the X/Open Call Level Interface specification has been accepted as part of the ISO Call Level Interface International Standard (ISO/IEC 9075-3:1995 SQL/CLI).

Microsoft developed a callable SQL interface called Open Database Connectivity (ODBC) for Microsoft operating systems based on a preliminary draft of X/Open CLI.

The ODBC specification also includes an operating environment where database-specific ODBC drivers are dynamically loaded at run time by a driver manager based on the data source (database name) provided on the connect request. The application is linked directly to a single driver manager library rather than to each DBMS’s library. The driver manager mediates the application’s function calls at run time and ensures they are directed to the appropriate DBMS-specific ODBC driver. Because the ODBC driver manager only knows about the ODBC-specific functions, DBMS-specific functions cannot be accessed in an ODBC environment. DBMS-specific dynamic SQL statements are supported through a mechanism called an escape clause.

ODBC is not limited to Microsoft operating systems; other implementations are available on various platforms.

The DB2 CLI load library can be loaded as an ODBC driver by an ODBC driver manager. For ODBC application development, you must obtain an ODBC Software Development Kit. For the Windows platform, the ODBC SDK is available as part of the Microsoft Data Access Components (MDAC) SDK, available for download from http://www.microsoft.com/data/. For non-Windows platforms, the ODBC SDK is provided by other vendors. When developing ODBC applications that may connect to DB2 servers, use the Call Level Interface Guide and Reference, Volume 1 and the Call Level Interface Guide and Reference, Volume 2 (for information on DB2 specific extensions and diagnostic information), in conjunction with the ODBC Programmer’s Reference and SDK Guide available from Microsoft.
Applications written directly to DB2 CLI link directly to the DB2 CLI load library. DB2 CLI includes support for many ODBC and ISO SQL/CLI functions, as well as DB2 specific functions.

The following DB2 features are available to both ODBC and DB2 CLI applications:
- double byte (graphic) data types
- stored procedures
- Distributed Unit of Work (DUOW), two phase commit
- compound SQL
- user defined types (UDT)
- user defined functions (UDF)

**Java application development for IBM data servers**

The DB2 and IBM Informix® Dynamic Server (IDS) database systems provide driver support for client applications and applets that are written in Java™.

You can access data in DB2 and IDS database systems using JDBC, SQL, or pureQuery.

**JDBC**

JDBC is an application programming interface (API) that Java applications use to access relational databases. IBM data server support for JDBC lets you write Java applications that access local DB2 or IDS data or remote relational data on a server that supports DRDA®.

**SQLJ**

SQLJ provides support for embedded static SQL in Java applications. SQLJ was initially developed by IBM, Oracle, and Tandem to complement the dynamic SQL JDBC model with a static SQL model.

For connections to DB2, in general, Java applications use JDBC for dynamic SQL and SQLJ for static SQL.

For connections to IDS, SQL statements in JDBC or SQLJ applications run dynamically.

Because SQLJ can inter-operate with JDBC, an application program can use JDBC and SQLJ within the same unit of work.

**pureQuery**

pureQuery is a high-performance data access platform that makes it easier to develop, optimize, secure, and manage data access. It consists of:
- Application programming interfaces that are built for ease of use and for simplifying the use of best practices
- Development tools, which are delivered in IBM Optim Development Studio, for Java and SQL development
- A runtime, which is delivered in IBM Optim pureQuery Runtime, for optimizing and securing database access and simplifying management tasks
With pureQuery, you can write Java applications that treat relational data as objects, whether that data is in databases or JDBC DataSource objects. Your applications can also treat objects that are stored in in-memory Java collections as though those objects are relational data. To query or update your relational data or Java objects, you use SQL.

For more information on pureQuery, see the Integrated Data Management Information Center.

**Schemas**

A *schema* is a collection of named objects; it provides a way to group those objects logically. A schema is also a name qualifier; it provides a way to use the same natural name for several objects, and to prevent ambiguous references to those objects.

For example, the schema names ‘INTERNAL’ and ‘EXTERNAL’ make it easy to distinguish two different SALES tables (INTERNAL.SALES, EXTERNAL.SALES).

Schemas also enable multiple applications to store data in a single database without encountering namespace collisions.

A schema is distinct from, and should not be confused with, an *XML schema*, which is a standard that describes the structure and validates the content of XML documents.

A schema can contain tables, views, nicknames, triggers, functions, packages, and other objects. A schema is itself a database object. It is explicitly created using the CREATE SCHEMA statement, with the current user or a specified authorization ID recorded as the schema owner. It can also be implicitly created when another object is created, if the user has IMPLICIT_SCHEMA authority.

A *schema name* is used as the high order part of a two-part object name. If the object is specifically qualified with a schema name when created, the object is assigned to that schema. If no schema name is specified when the object is created, the default schema name is used (specified in the CURRENT SCHEMA special register).

For example, a user with DBADM authority creates a schema called C for user A:

```sql
CREATE SCHEMA C AUTHORIZATION A
```

User A can then issue the following statement to create a table called X in schema C (provided that user A has the CREATETAB database authority):

```sql
CREATE TABLE C.X (COL1 INT)
```

Some schema names are reserved. For example, built-in functions belong to the SYSIBM schema, and the pre-installed user-defined functions belong to the SYSFUN schema.

When a database is created, if it is not created with the RESTRICTIVE option, all users have IMPLICIT_SCHEMA authority. With this authority, users implicitly create a schema whenever they create an object with a schema name that does not already exist. When schemas are implicitly created, CREATEIN privileges are granted which allows any user to create other objects in this schema. The ability to create objects such as aliases, distinct types, functions, and triggers is extended to
Schemas

implicitly-created schemas. The default privileges on an implicitly-created schema provide backward compatibility with previous versions.

If IMPLICIT_SCHEMA authority is revoked from PUBLIC, schemas can be explicitly created using the CREATE SCHEMA statement, or implicitly created by users (such as those with DBADM authority) who have been granted IMPLICIT_SCHEMA authority. Although revoking IMPLICIT_SCHEMA authority from PUBLIC increases control over the use of schema names, it can result in authorization errors when existing applications attempt to create objects.

Schemas also have privileges, allowing the schema owner to control which users have the privilege to create, alter, copy, and drop objects in the schema. This provides a way to control the manipulation of a subset of objects in the database. A schema owner is initially given all of these privileges on the schema, with the ability to grant the privileges to others. An implicitly-created schema is owned by the system, and all users are initially given the privilege to create objects in such a schema. A user with ACCESSCTRL or SECADM authority can change the privileges that are held by users on any schema. Therefore, access to create, alter, copy, and drop objects in any schema (even one that was implicitly created) can be controlled.

Tables

Tables are logical structures maintained by the database manager. Tables are made up of columns and rows.

At the intersection of every column and row is a specific data item called a value. A column is a set of values of the same type or one of its subtypes. A row is a sequence of values arranged so that the nth value is a value of the nth column of the table.

An application program can determine the order in which the rows are populated into the table, but the actual order of rows is determined by the database manager, and typically cannot be controlled. Multidimensional clustering (MDC) provides some sense of clustering, but not actual ordering between the rows.

Types of tables

DB2 databases store data in tables. In addition to tables used to store persistent data, there are also tables that are used for presenting results, summary tables and temporary tables; multidimensional clustering tables offer specific advantages in a warehouse environment, whereas partitioned tables let you spread data across more than one database partition.

Base tables

These types of tables hold persistent data. There are different kinds of base tables, including

Regular tables

Regular tables with indexes are the “general purpose” table choice.

Multidimensional clustering (MDC) tables

These types of tables are implemented as tables that are physically clustered on more than one key, or dimension, at the same time. MDC tables are used in data warehousing and large database environments. Clustering indexes on regular tables support single-dimensional clustering of data. MDC tables provide the benefits of data clustering across more than one dimension. MDC
Types of tables

tables provide guaranteed clustering within the composite dimensions. By contrast, although you can have a clustered index with regular tables, clustering in this case is attempted by the database manager, but not guaranteed and it typically degrades over time. MDC tables can coexist with partitioned tables and can themselves be partitioned tables.

Range-clustered tables (RCT)
These types of tables are implemented as sequential clusters of data that provide fast, direct access. Each record in the table has a predetermined record ID (RID) which is an internal identifier used to locate a record in a table. RCT tables are used where the data is tightly clustered across one or more columns in the table. The largest and smallest values in the columns define the range of possible values. You use these columns to access records in the table; this is the most optimal method of utilizing the predetermined record identifier (RID) aspect of RCT tables.

Temporary tables
These types of tables are used as temporary work tables for a variety of database operations. Declared temporary tables (DGTs) do not appear in the system catalog, which makes them not persistent for use by, and not able to be shared with other applications. When the application using this table terminates or disconnects from the database, any data in the table is deleted and the table is dropped. By contrast, created temporary tables (CGTs) do appear in the system catalog and are not required to be defined in every session where they are used. As a result, they are persistent and able to be shared with other applications across different connections.

Neither type of temporary table supports
- User-defined reference or user-defined structured type columns
- LONG VARCHAR columns

In addition XML columns cannot be used in created temporary tables.

Materialized query tables
These types of tables are defined by a query that is also used to determine the data in the table. Materialized query tables can be used to improve the performance of queries. If the database manager determines that a portion of a query can be resolved using a summary table, the database manager can rewrite the query to use the summary table. This decision is based on database configuration settings, such as the CURRENT REFRESH AGE and the CURRENT QUERY OPTIMIZATION special registers. A summary table is a specialized type of materialized query table.

You can create all of the preceding types of tables using the CREATE TABLE statement.

Depending on what your data is going to look like, you might find one table type offers specific capabilities that can optimize storage and query performance. For example, if you have data records that will be loosely clustered (not monotonically increasing), consider using a regular table and indexes. If you have data records that will have duplicate (but not unique) values in the key, you should not use a range-clustered table. Also, if you cannot afford to preallocate a fixed amount of storage on disk for the range-clustered tables you might want, you should not use this type of table. If you have data that has the potential for being clustered along
Types of tables

multiple dimensions, such as a table tracking retail sales by geographic region, division and supplier, a multidimensional clustering table might suit your purposes.

In addition to the various table types described above, you also have options for such characteristics as partitioning, which can improve performance for tasks such as rolling in table data. Partitioned tables can also hold much more information than a regular, nonpartitioned table. You can also exploit capabilities such as compression, which can help you significantly reduce your data storage costs.

Constraints

Within any business, data must often adhere to certain restrictions or rules. For example, an employee number must be unique. The database manager provides constraints as a way to enforce such rules.

The following types of constraints are available:

- NOT NULL constraints
- Unique (or unique key) constraints
- Primary key constraints
- Foreign key (or referential integrity) constraints
- (Table) Check constraints
- Informational constraints

Constraints are only associated with tables and are either defined as part of the table creation process (using the CREATE TABLE statement) or are added to a table’s definition after the table has been created (using the ALTER TABLE statement). You can use the ALTER TABLE statement to modify constraints. In most cases, existing constraints can be dropped at any time; this action does not affect the table’s structure or the data stored in it.

Note: Unique and primary constraints are only associated with table objects, they are often enforced through the use of one or more unique or primary key indexes.

Indexes

An index is a set of pointers that are logically ordered by the values of one or more keys. The pointers can refer to rows in a table, blocks in an MDC table, XML data in an XML storage object, and so on.

Indexes are used to:

- Improve performance. In most cases, access to data is faster with an index.
  Although an index cannot be created for a view, an index created for the table on which a view is based can sometimes improve the performance of operations on that view.
- Ensure uniqueness. A table with a unique index cannot have rows with identical keys.

As data is added to a table, it is appended to the bottom (unless other actions have been carried out on the table or the data being added). There is no inherent order to the data. When searching for a particular row of data, each row of the table from first to last must be checked. Indexes are used as a means to access the data within the table in an order that might otherwise not be available.
Typically, when you search for data in a table, you are looking for rows with columns that have specific values. A column value in a row of data can be used to identify the entire row. For example, an employee number would probably uniquely define a specific individual employee. Or, more than one column might be needed to identify the row. For example, a combination of customer name and telephone number. Columns in an index used to identify data rows are known as keys. A column can be used in more than one key.

An index is ordered by the values within a key. Keys can be unique or non-unique. Each table should have at least one unique key; but can also have other, non-unique keys. Each index has exactly one key. For example, you might use the employee ID number (unique) as the key for one index and the department number (non-unique) as the key for a different index.

Not all indexes point to rows in a table. MDC block indexes point to extents (or blocks) of the data. XML indexes for XML data use particular XML pattern expressions to index paths and values in XML documents stored within a single column. The data type of that column must be XML. Both MDC block indexes and XML indexes are system generated indexes.

Example

Table A in Figure 1 has an index based on the employee numbers in the table. This key value provides a pointer to the rows in the table. For example, employee number 19 points to employee KMP. An index allows efficient access to rows in a table by creating a path to the data through pointers.

Unique indexes can be created to ensure uniqueness of the index key. An index key is a column or an ordered collection of columns on which an index is defined. Using a unique index will ensure that the value of each index key in the indexed column or columns is unique.

Figure 1 shows the relationship between an index and a table.

Database

![Index A](image1.png)

![Table A](image2.png)

Figure 1. Relationship between an index and a table

Figure 2 on page 10 illustrates the relationships among some database objects. It also shows that tables, indexes, and long data are stored in table spaces.
Triggers

A trigger defines a set of actions that are performed in response to an insert, update, or delete operation on a specified table. When such an SQL operation is executed, the trigger is said to have been activated. Triggers are optional and are defined using the CREATE TRIGGER statement.

Triggers can be used, along with referential constraints and check constraints, to enforce data integrity rules. Triggers can also be used to cause updates to other tables, automatically generate or transform values for inserted or updated rows, or invoke functions to perform tasks such as issuing alerts.

Triggers are a useful mechanism for defining and enforcing transitional business rules, which are rules that involve different states of the data (for example, a salary that cannot be increased by more than 10 percent).

Using triggers places the logic that enforces business rules inside the database. This means that applications are not responsible for enforcing these rules. Centralized logic that is enforced on all of the tables means easier maintenance, because changes to application programs are not required when the logic changes.

The following are specified when creating a trigger:

- The subject table specifies the table for which the trigger is defined.
- The trigger event defines a specific SQL operation that modifies the subject table. The event can be an insert, update, or delete operation.
- The trigger activation time specifies whether the trigger should be activated before or after the trigger event occurs.

The statement that causes a trigger to be activated includes a set of affected rows. These are the rows of the subject table that are being inserted, updated, or deleted. The trigger granularity specifies whether the actions of the trigger are performed once for the statement or once for each of the affected rows.
The triggered action consists of an optional search condition and a set of statements that are executed whenever the trigger is activated. The statements are only executed if the search condition evaluates to true. If the trigger activation time is before the trigger event, triggered actions can include statements that select, set transition variables, or signal SQL states. If the trigger activation time is after the trigger event, triggered actions can include statements that select, insert, update, delete, or signal SQL states.

The triggered action can refer to the values in the set of affected rows using transition variables. Transition variables use the names of the columns in the subject table, qualified by a specified name that identifies whether the reference is to the old value (before the update) or the new value (after the update). The new value can also be changed using the SET Variable statement in before, insert, or update triggers.

Another means of referring to the values in the set of affected rows is to use transition tables. Transition tables also use the names of the columns in the subject table, but specify a name to allow the complete set of affected rows to be treated as a table. Transition tables can only be used in AFTER triggers (that is, not with BEFORE and INSTEAD OF triggers), and separate transition tables can be defined for old and new values.

Multiple triggers can be specified for a combination of table, event (INSERT, UPDATE, DELETE), or activation time (BEFORE, AFTER, INSTEAD OF). When more than one trigger exists for a particular table, event, and activation time, the order in which the triggers are activated is the same as the order in which they were created. Thus, the most recently created trigger is the last trigger to be activated.

The activation of a trigger might cause trigger cascading, which is the result of the activation of one trigger that executes statements that cause the activation of other triggers or even the same trigger again. The triggered actions might also cause updates resulting from the application of referential integrity rules for deletions that can, in turn, result in the activation of additional triggers. With trigger cascading, a chain of triggers and referential integrity delete rules can be activated, causing significant change to the database as a result of a single INSERT, UPDATE, or DELETE statement.

When multiple triggers have insert, update, or delete actions against the same object, conflict resolution mechanism, like temporary tables, are used to resolve access conflicts, and this can have a noticeable impact on performance, particularly in partitioned database environments.

Views

A view is an efficient way of representing data without the need to maintain it. A view is not an actual table and requires no permanent storage. A “virtual table” is created and used.

A view provides a different way of looking at the data in one or more tables; it is a named specification of a result table. The specification is a SELECT statement that is run whenever the view is referenced in an SQL statement. A view has columns and rows just like a table. All views can be used just like tables for data retrieval. Whether a view can be used in an insert, update, or delete operation depends on its definition.
Views

A view can include all or some of the columns or rows contained in the tables on which it is based. For example, you can join a department table and an employee table in a view, so that you can list all employees in a particular department.

Figure 3 shows the relationship between tables and views.

You can use views to control access to sensitive data, because views allow multiple users to see different presentations of the same data. For example, several users might be accessing a table of data about employees. A manager sees data about his or her employees but not employees in another department. A recruitment officer sees the hire dates of all employees, but not their salaries; a financial officer sees the salaries, but not the hire dates. Each of these users works with a view derived from the table. Each view appears to be a table and has its own name.

When the column of a view is directly derived from the column of a base table, that view column inherits any constraints that apply to the table column. For example, if a view includes a foreign key of its table, insert and update operations using that view are subject to the same referential constraints as is the table. Also, if the table of a view is a parent table, delete and update operations using that view are subject to the same rules as are delete and update operations on the table.

A view can derive the data type of each column from the result table, or base the types on the attributes of a user-defined structured type. This is called a typed view. Similar to a typed table, a typed view can be part of a view hierarchy. A subview inherits columns from its superview. The term subview applies to a typed view and to all typed views that are below it in the view hierarchy. A proper subview of a view V is a view below V in the typed view hierarchy.

A view can become inoperative (for example, if the table is dropped); if this occurs, the view is no longer available for SQL operations.
**Aliases**

An *alias* is an alternative name for an object such as a module, table or another alias. It can be used to reference an object wherever that object can be referenced directly.

An alias cannot be used in all contexts; for example, it cannot be used in the check condition of a check constraint. An alias cannot reference a declared temporary table but it can reference a created temporary table.

Like other objects, an alias can be created, dropped, and have comments associated with it. Aliases can refer to other aliases in a process called *chaining* as long as there are no circular references. Aliases do not require any special authority or privilege to use them. Access to the object referred to by an alias, however, does require the authorization associated with that object.

If an alias is defined as a *public* alias, it can be referenced by its unqualified name without any impact from the current default schema name. It can also be referenced using the qualifier SYSPUBLIC.

*Synonym* is an alternative name for alias.

For more information, refer to "Aliases in identifiers" in the SQL Reference, Volume 1.

---

**Package**

A *package* is an object stored in the database that includes the information needed to process the SQL statements associated with one source file of an application program.

It is generated by either:

- Precompiling a source file with the PREP command
- Binding a bind file that was generated by the precompiler with the BIND command.

---

**Authorization, privileges, and object ownership**

Users (identified by an authorization ID) can successfully execute operations only if they have the authority to perform the specified function. To create a table, a user must be authorized to create tables; to alter a table, a user must be authorized to alter the table; and so forth.

The database manager requires that each user be specifically authorized to use each database function needed to perform a specific task. A user can acquire the necessary authorization through a grant of that authorization to their user ID or through membership in a role or a group that holds that authorization.

There are three forms of authorization, *administrative authority*, *privileges*, and *LBAC credentials*. In addition, ownership of objects brings with it a degree of authorization on the objects created. These forms of authorization are discussed below.
Authorization, privileges, and object ownership

Administrative authority

The person or persons holding administrative authority are charged with the task of controlling the database manager and are responsible for the safety and integrity of the data.

System-level authorization

The system-level authorities provide varying degrees of control over instance-level functions:

- SYSADM (system administrator) authority
  The SYSADM (system administrator) authority provides control over all the resources created and maintained by the database manager. The system administrator possesses all the authorities of SYSCTRL, SYSMAINT, and SYSMON authority. The user who has SYSADM authority is responsible both for controlling the database manager, and for ensuring the safety and integrity of the data.

- SYSCTRL authority
  The SYSCTRL authority provides control over operations that affect system resources. For example, a user with SYSCTRL authority can create, update, start, stop, or drop a database. This user can also start or stop an instance, but cannot access table data. Users with SYSCTRL authority also have SYSMON authority.

- SYSMAINT authority
  The SYSMAINT authority provides the authority required to perform maintenance operations on all databases associated with an instance. A user with SYSMAINT authority can update the database configuration, backup a database or table space, restore an existing database, and monitor a database. Like SYSCTRL, SYSMAINT does not provide access to table data. Users with SYSMAINT authority also have SYSMON authority.

- SYSMON (system monitor) authority
  The SYSMON (system monitor) authority provides the authority required to use the database system monitor.

Database-level authorization

The database level authorities provide control within the database:

- DBADM (database administrator)
  The DBADM authority level provides administrative authority over a single database. This database administrator possesses the privileges required to create objects and issue database commands.
  The DBADM authority can only be granted by a user with SECADM authority. The DBADM authority cannot be granted to PUBLIC.

- SECADM (security administrator)
  The SECADM authority level provides administrative authority for security over a single database. The security administrator authority possesses the ability to manage database security objects (database roles, audit policies, trusted contexts, security label components, and security labels) and grant and revoke all database privileges and authorities. A user with SECADM authority can transfer the ownership of objects that they do not own. They can also use the AUDIT statement to associate an audit policy with a particular database or database object at the server.
Authorization, privileges, and object ownership

The SECADM authority has no inherent privilege to access data stored in tables. It can only be granted by a user with SECADM authority. The SECADM authority cannot be granted to PUBLIC.

- **SQLADM (SQL administrator)**
  The SQLADM authority level provides administrative authority to monitor and tune SQL statements within a single database. It can be granted by a user with ACCESSCTRL or SECADM authority.

- **WLMADM (workload management administrator)**
  The WLMADM authority provides administrative authority to manage workload management objects, such as service classes, work action sets, work class sets, and workloads. It can be granted by a user with ACCESSCTRL or SECADM authority.

- **EXPLAIN (explain authority)**
  The EXPLAIN authority level provides administrative authority to explain query plans without gaining access to data. It can only be granted by a user with ACCESSCTRL or SECADM authority.

- **ACCESSCTRL (access control authority)**
  The ACCESSCTRL authority level provides administrative authority to issue the following GRANT (and REVOKE) statements. ACCESSCTRL authority can only be granted by a user with SECADM authority. The ACCESSCTRL authority cannot be granted to PUBLIC.
    - **GRANT (Database Authorities)**
      ACCESSCTRL authority does not give the holder the ability to grant ACCESSCTRL, DATAACCESS, DBADM, or SECADM authority. Only a user who has SECADM authority can grant these authorities.
    - **GRANT (Global Variable Privileges)**
    - **GRANT (Index Privileges)**
    - **GRANT (Module Privileges)**
    - **GRANT (Package Privileges)**
    - **GRANT (Routine Privileges)**
    - **GRANT (Schema Privileges)**
    - **GRANT (Sequence Privileges)**
    - **GRANT (Server Privileges)**
    - **GRANT (Table, View, or Nickname Privileges)**
    - **GRANT (Table Space Privileges)**
    - **GRANT (Workload Privileges)**
    - **GRANT (XSR Object Privileges)**

- **DATAACCESS (data access authority)**
  The DATAACCESS authority level provides the following privileges and authorities. It can be granted only by a user who holds SECADM authority. The DATAACCESS authority cannot be granted to PUBLIC.
    - **LOAD authority**
    - **SELECT, INSERT, UPDATE, DELETE privilege on tables, views, nicknames, and materialized query tables**
    - **EXECUTE privilege on packages**
    - **EXECUTE privilege on modules**
    - **EXECUTE privilege on routines**
Authorization, privileges, and object ownership

Except on the audit routines: AUDIT_ARCHIVE, AUDIT_LIST_LOGS, AUDIT_DELIM_EXTRACT.

• Database authorities (non-administrative)

To perform activities such as creating a table or a routine, or for loading data into a table, specific database authorities are required. For example, the LOAD database authority is required for use of the load utility to load data into tables (a user must also have INSERT privilege on the table).

Privileges

A privilege is a permission to perform an action or a task. Authorized users can create objects, have access to objects they own, and can pass on privileges on their own objects to other users by using the GRANT statement.

Privileges may be granted to individual users, to groups, or to PUBLIC. PUBLIC is a special group that consists of all users, including future users. Users that are members of a group will indirectly take advantage of the privileges granted to the group, where groups are supported.

The CONTROL privilege: Possessing the CONTROL privilege on an object allows a user to access that database object, and to grant and revoke privileges to or from other users on that object.

Note: The CONTROL privilege only applies to tables, views, nicknames, indexes, and packages.

If a different user requires the CONTROL privilege to that object, a user with SECADM or ACCESSCTRL authority could grant the CONTROL privilege to that object. The CONTROL privilege cannot be revoked from the object owner, however, the object owner can be changed by using the TRANSFER OWNERSHIP statement.

Individual privileges: Individual privileges can be granted to allow a user to carry out specific tasks on specific objects. Users with the administrative authorities ACCESSCTRL or SECADM, or with the CONTROL privilege, can grant and revoke privileges to and from users.

Individual privileges and database authorities allow a specific function, but do not include the right to grant the same privileges or authorities to other users. The right to grant table, view, schema, package, routine, and sequence privileges to others can be extended to other users through the WITH GRANT OPTION on the GRANT statement. However, the WITH GRANT OPTION does not allow the person granting the privilege to revoke the privilege once granted. You must have SECADM authority, ACCESSCTRL authority, or the CONTROL privilege to revoke the privilege.

Privileges on objects in a package or routine: When a user has the privilege to execute a package or routine, they do not necessarily require specific privileges on the objects used in the package or routine. If the package or routine contains static SQL or XQuery statements, the privileges of the owner of the package are used for those statements. If the package or routine contains dynamic SQL or XQuery statements, the authorization ID used for privilege checking depends on the setting of the DYNAMICRULES BIND option of the package issuing the dynamic query statements, and whether those statements are issued when the package is being used in the context of a routine.
Authorization, privileges, and object ownership

A user or group can be authorized for any combination of individual privileges or authorities. When a privilege is associated with an object, that object must exist. For example, a user cannot be given the SELECT privilege on a table unless that table has previously been created.

**Note:** Care must be taken when an authorization name representing a user or a group is granted authorities and privileges and there is no user, or group created with that name. At some later time, a user or a group can be created with that name and automatically receive all of the authorities and privileges associated with that authorization name.

The REVOKE statement is used to revoke previously granted privileges. The revoking of a privilege from an authorization name revokes the privilege granted by all authorization names.

Revoking a privilege from an authorization name does not revoke that same privilege from any other authorization names that were granted the privilege by that authorization name. For example, assume that CLAIRE grants SELECT WITH GRANT OPTION to RICK, then RICK grants SELECT to BOBBY and CHRIS. If CLAIRE revokes the SELECT privilege from RICK, BOBBY and CHRIS still retain the SELECT privilege.

**LBAC credentials**

Label-based access control (LBAC) lets the security administrator decide exactly who has write access and who has read access to individual rows and individual columns. The security administrator configures the LBAC system by creating security policies. A security policy describes the criteria used to decide who has access to what data. Only one security policy can be used to protect any one table but different tables can be protected by different security policies.

After creating a security policy, the security administrator creates database objects, called security labels and exemptions that are part of that policy. A security label describes a certain set of security criteria. An exemption allows a rule for comparing security labels not to be enforced for the user who holds the exemption, when they access data protected by that security policy.

Once created, a security label can be associated with individual columns and rows in a table to protect the data held there. Data that is protected by a security label is called protected data. A security administrator allows users access to protected data by granting them security labels. When a user tries to access protected data, that user’s security label is compared to the security label protecting the data. The protecting label blocks some security labels and does not block others.

**Object ownership**

When an object is created, one authorization ID is assigned *ownership* of the object. Ownership means the user is authorized to reference the object in any applicable SQL or XQuery statement.

When an object is created within a schema, the authorization ID of the statement must have the required privilege to create objects in the implicitly or explicitly specified schema. That is, the authorization name must either be the owner of the schema, or possess the CREATEIN privilege on the schema.
Authorization, privileges, and object ownership

Note: This requirement is not applicable when creating table spaces, buffer pools or database partition groups. These objects are not created in schemas.

When an object is created, the authorization ID of the statement is the definer of that object and by default becomes the owner of the object after it is created.

Note: One exception exists. If the AUTHORIZATION option is specified for the CREATE SCHEMA statement, any other object that is created as part of the CREATE SCHEMA operation is owned by the authorization ID specified by the AUTHORIZATION option. Any objects that are created in the schema after the initial CREATE SCHEMA operation, however, are owned by the authorization ID associated with the specific CREATE statement.

For example, the statement CREATE SCHEMA SCOTTSTUFF AUTHORIZATION SCOTT
CREATE TABLE T1 (C1 INT) creates the schema SCOTTSTUFF and the table SCOTTSTUFF.T1, which are both owned by SCOTT. Assume that the user BOBBY is granted the CREATEIN privilege on the SCOTTSTUFF schema and creates an index on the SCOTTSTUFF.T1 table. Because the index is created after the schema, BOBBY owns the index on SCOTTSTUFF.T1.

Privileges are assigned to the object owner based on the type of object being created:

- The CONTROL privilege is implicitly granted on newly created tables, indexes, and packages. This privilege allows the object creator to access the database object, and to grant and revoke privileges to or from other users on that object. If a different user requires the CONTROL privilege to that object, a user with ACCESSCTRL or SECADM authority must grant the CONTROL privilege to that object. The CONTROL privilege cannot be revoked by the object owner.
- The CONTROL privilege is implicitly granted on newly created views if the object owner has the CONTROL privilege on all the tables, views, and nicknames referenced by the view definition.
- Other objects like triggers, routines, sequences, table spaces, and buffer pools do not have a CONTROL privilege associated with them. The object owner does, however, automatically receive each of the privileges associated with the object and those privileges are with the WITH GRANT OPTION, where supported. Therefore the object owner can provide these privileges to other users by using the GRANT statement. For example, if USER1 creates a table space, USER1 automatically has the USEAUTH privilege with the WITH GRANT OPTION on this table space and can grant the USEAUTH privilege to other users. In addition, the object owner can alter, add a comment on, or drop the object. These authorizations are implicit for the object owner and cannot be revoked.

Certain privileges on the object, such as altering a table, can be granted by the owner, and can be revoked from the owner by a user who has ACCESSCTRL or SECADM authority. Certain privileges on the object, such as commenting on a table, cannot be granted by the owner and cannot be revoked from the owner. Use the TRANSFER OWNERSHIP statement to move these privileges to another user. When an object is created, the authorization ID of the statement is the definer of that object and by default becomes the owner of the object after it is created. However, when you use the BIND command to create a package and you specify the OWNER authorization id option, the owner of objects created by the static SQL statements in the package is the value of authorization id. In addition, if the AUTHORIZATION clause is specified on a CREATE SCHEMA statement, the authorization name specified after the AUTHORIZATION keyword is the owner of the schema.
Authorization, privileges, and object ownership

A security administrator or the object owner can use the TRANSFER OWNERSHIP statement to change the ownership of a database object. An administrator can therefore create an object on behalf of an authorization ID, by creating the object using the authorization ID as the qualifier, and then using the TRANSFER OWNERSHIP statement to transfer the ownership that the administrator has on the object to the authorization ID.

System catalog views

The database manager maintains a set of tables and views that contain information about the data under its control. These tables and views are collectively known as the system catalog.

The system catalog contains information about the logical and physical structure of database objects such as tables, views, indexes, packages, and functions. It also contains statistical information. The database manager ensures that the descriptions in the system catalog are always accurate.

The system catalog views are like any other database view. SQL statements can be used to query the data in the system catalog views. A set of updatable system catalog views can be used to modify certain values in the system catalog.

Application processes, concurrency, and recovery

All SQL programs execute as part of an application process or agent. An application process involves the execution of one or more programs, and is the unit to which the database manager allocates resources and locks. Different application processes might involve the execution of different programs, or different executions of the same program.

More than one application process can request access to the same data at the same time. Locking is the mechanism that is used to maintain data integrity under such conditions, preventing, for example, two application processes from updating the same row of data simultaneously.

The database manager acquires locks to prevent uncommitted changes made by one application process from being accidentally perceived by any other process. The database manager releases all locks it has acquired and retained on behalf of an application process when that process ends. However, an application process can explicitly request that locks be released sooner. This is done using a commit operation, which releases locks that were acquired during a unit of work and also commits database changes that were made during the unit of work.

A unit of work (UOW) is a recoverable sequence of operations within an application process. A unit of work is initiated when an application process starts, or when the previous UOW ends because of something other than the termination of the application process. A unit of work ends with a commit operation, a rollback operation, or the end of an application process. A commit or rollback operation affects only the database changes that were made within the UOW that is ending.

The database manager provides a means of backing out of uncommitted changes that were made by an application process. This might be necessary in the event of a failure on the part of an application process, or in the case of a deadlock or lock timeout situation. An application process can explicitly request that its database changes be cancelled. This is done using a rollback operation.
As long as these changes remain uncommitted, other application processes are unable to see them, and the changes can be rolled back. This is not true, however, if the prevailing isolation level is uncommitted read (UR). After they are committed, these database changes are accessible to other application processes and can no longer be rolled back.

Both DB2 call level interface (CLI) and embedded SQL allow for a connection mode called concurrent transactions, which supports multiple connections, each of which is an independent transaction. An application can have multiple concurrent connections to the same database.

Locks that are acquired by the database manager on behalf of an application process are held until the end of a UOW, except when the isolation level is cursor stability (CS, in which the lock is released as the cursor moves from row to row) or uncommitted read (UR).

An application process is never prevented from performing operations because of its own locks. However, if an application uses concurrent transactions, the locks from one transaction might affect the operation of a concurrent transaction.

The initiation and the termination of a UOW define points of consistency within an application process. For example, a banking transaction might involve the transfer of funds from one account to another. Such a transaction would require that these funds be subtracted from the first account, and then added to the second account. Following the subtraction step, the data is inconsistent. Only after the funds have been added to the second account is consistency reestablished. When both steps are complete, the commit operation can be used to end the UOW, thereby making the changes available to other application processes. If a failure occurs before the UOW ends, the database manager will roll back any uncommitted changes to restore data consistency.

Figure 4. Unit of work with a COMMIT statement
Isolation levels

The isolation level that is associated with an application process determines the degree to which the data that is being accessed by that process is locked or isolated from other concurrently executing processes. The isolation level is in effect for the duration of a unit of work.

The isolation level of an application process therefore specifies:
- The degree to which rows that are read or updated by the application are available to other concurrently executing application processes
- The degree to which the update activity of other concurrently executing application processes can affect the application

The isolation level for static SQL statements is specified as an attribute of a package and applies to the application processes that use that package. The isolation level is specified during the program preparation process by setting the ISOLATION bind or precompile option. For dynamic SQL statements, the default isolation level is the isolation level that was specified for the package preparing the statement. Use the SET CURRENT ISOLATION statement to specify a different isolation level for dynamic SQL statements that are issued within a session. For more information, see “CURRENT ISOLATION special register”. For both static SQL statements and dynamic SQL statements, the isolation-clause in a select-statement overrides both the special register (if set) and the bind option value. For more information, see “Select-statement”.

Isolation levels are enforced by locks, and the type of lock that is used limits or prevents access to the data by concurrent application processes. Declared temporary tables and their rows cannot be locked because they are only accessible to the application that declared them.

The database manager supports three general categories of locks:

**Share (S)**
- Under a S lock, concurrent application processes are limited to read-only operations on the data.

**Update (U)**
- Under a U lock, concurrent application processes are limited to read-only
Isolation levels

operations on the data, if these processes have not declared that they might
update a row. The database manager assumes that the process currently
looking at a row might update it.

Exclusive (X)
Under an X lock, concurrent application processes are prevented from
accessing the data in any way. This does not apply to application processes
with an isolation level of uncommitted read (UR), which can read but not modify the data.

Regardless of the isolation level, the database manager places exclusive locks on
every row that is inserted, updated, or deleted. Thus, all isolation levels ensure
that any row that is changed by an application process during a unit of work is
not changed by any other application process until the unit of work is complete.

The database manager supports four isolation levels.

  • “Repeatable read (RR)”
  • “Read stability (RS)” on page 23
  • “Cursor stability (CS)” on page 23
  • “Uncommitted read (UR)” on page 24

Note: Some host database servers support the no commit (NC) isolation level. On
other database servers, this isolation level behaves like the uncommitted read isolation level.

A detailed description of each isolation level follows, in decreasing order of
performance impact, but in increasing order of the care that is required when
accessing or updating data.

Repeatable read (RR)
The repeatable read isolation level locks all the rows that an application references
during a unit of work (UOW). If an application issues a SELECT statement twice
within the same unit of work, the same result is returned each time. Under RR,
lost updates, access to uncommitted data, non-repeatable reads, and phantom
reads are not possible.

Under RR, an application can retrieve and operate on the rows as many times as
necessary until the UOW completes. However, no other application can update,
delete, or insert a row that would affect the result set until the UOW completes.
Applications running under the RR isolation level cannot see the uncommitted changes of other applications. This isolation level ensures that all returned data
remains unchanged until the time the application sees the data, even when
temporary tables or row blocking is used.

Every referenced row is locked, not just the rows that are retrieved. For example, if
you scan 10,000 rows and apply predicates to them, locks are held on all 10,000
rows, even if, say, only 10 rows qualify. Another application cannot insert or
update a row that would be added to the list of rows referenced by a query if that
query were to be executed again. This prevents phantom reads.

Because RR can acquire a considerable number of locks, this number might exceed
limits specified by the locklist and maxlocks database configuration parameters.
To avoid lock escalation, the optimizer might elect to acquire a single table-level
lock for an index scan, if it appears that lock escalation is likely. If you do not want table-level locking, use the read stability isolation level.

While evaluating referential constraints, the DB2 server might occasionally upgrade the isolation level used on scans of the foreign table to RR, regardless of the isolation level that was previously set by the user. This results in additional locks being held until commit time, which increases the likelihood of a deadlock or a lock timeout. To avoid these problems, create an index that contains only the foreign key columns, and which the referential integrity scan can use instead.

**Read stability (RS)**

The *read stability* isolation level locks only those rows that an application retrieves during a unit of work. RS ensures that any qualifying row read during a UOW cannot be changed by other application processes until the UOW completes, and that any row changed by another application process cannot be read until the change is committed by that process. Under RS, access to uncommitted data and non-repeatable reads are not possible. However, phantom reads are possible.

This isolation level ensures that all returned data remains unchanged until the time the application sees the data, even when temporary tables or row blocking is used.

The RS isolation level provides both a high degree of concurrency and a stable view of the data. To that end, the optimizer ensures that table-level locks are not obtained until lock escalation occurs.

The RS isolation level is suitable for an application that:

• Operates in a concurrent environment
• Requires qualifying rows to remain stable for the duration of a unit of work
• Does not issue the same query more than once during a unit of work, or does not require the same result set when a query is issued more than once during a unit of work

**Cursor stability (CS)**

The *cursor stability* isolation level locks any row being accessed during a transaction while the cursor is positioned on that row. This lock remains in effect until the next row is fetched or the transaction terminates. However, if any data in the row was changed, the lock is held until the change is committed.

Under this isolation level, no other application can update or delete a row while an updatable cursor is positioned on that row. Under CS, access to the uncommitted data of other applications is not possible. However, non-repeatable reads and phantom reads are possible.

CS is the default isolation level. It is suitable when you want maximum concurrency and need to see only committed data.

**Note:** Under the *currently committed* semantics introduced in Version 9.7, only committed data is returned, as was the case previously, but now readers do not wait for updaters to release row locks. Instead, readers return data that is based on the currently committed version; that is, data prior to the start of the write operation.
Uncommitted read (UR)

The *uncommitted read* isolation level allows an application to access the uncommitted changes of other transactions. Moreover, UR does not prevent another application from accessing a row that is being read, unless that application is attempting to alter or drop the table.

Under UR, access to uncommitted data, non-repeatable reads, and phantom reads are possible. This isolation level is suitable if you run queries against read-only tables, or if you issue SELECT statements only, and seeing data that has not been committed by other applications is not a problem.

UR works differently for read-only and updatable cursors.

- Read-only cursors can access most of the uncommitted changes of other transactions.
- Tables, views, and indexes that are being created or dropped by other transactions are not available while the transaction is processing. Any other changes by other transactions can be read before they are committed or rolled back. Updatable cursors operating under UR behave as though the isolation level were CS.

If an uncommitted read application uses ambiguous cursors, it might use the CS isolation level when it runs. The ambiguous cursors can be escalated to CS if the value of the BLOCKING option on the PREP or BIND command is UNAMBIG (the default). To prevent this escalation:

- Modify the cursors in the application program to be unambiguous. Change the SELECT statements to include the FOR READ ONLY clause.
- Let the cursors in the application program remain ambiguous, but precompile the program or bind it with the BLOCKING ALL and STATICREADONLY YES options to enable the ambiguous cursors to be treated as read-only when the program runs.

Comparison of isolation levels

Table 1 summarizes the supported isolation levels.

<table>
<thead>
<tr>
<th>Can an application see uncommitted changes made by other application processes?</th>
<th>UR</th>
<th>CS</th>
<th>RS</th>
<th>RR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Can an application update uncommitted changes made by other application processes?</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Can the re-execution of a statement be affected by other application processes?</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Can updated rows be updated by other application processes?</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Can updated rows be read by other application processes that are running at an isolation level other than UR?</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Can updated rows be read by other application processes that are running at the UR isolation level?</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Isolation levels

Table 1. Comparison of isolation levels (continued)

<table>
<thead>
<tr>
<th>Can accessed rows be updated by other application processes?</th>
<th>UR</th>
<th>CS</th>
<th>RS</th>
<th>RR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Can accessed rows be read by other application processes?</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Can the current row be updated or deleted by other application processes?</td>
<td>Yes/No</td>
<td>Yes/No</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

Note:
1. An example of the phantom read phenomenon is as follows: Unit of work UW1 reads the set of n rows that satisfies some search condition. Unit of work UW2 inserts one or more rows that satisfy the same search condition and then commits. If UW1 subsequently repeats its read with the same search condition, it sees a different result set: the rows that were read originally plus the rows that were inserted by UW2.
2. If your label-based access control (LBAC) credentials change between reads, results for the second read might be different because you have access to different rows.
3. The isolation level offers no protection to the application if the application is both reading from and writing to a table. For example, an application opens a cursor on a table and then performs an insert, update, or delete operation on the same table. The application might see inconsistent data when more rows are fetched from the open cursor.
4. An example of the non-repeatable read phenomenon is as follows: Unit of work UW1 reads a row. Unit of work UW2 modifies that row and commits. If UW1 subsequently reads that row again, it might see a different value.
5. An example of the dirty read phenomenon is as follows: Unit of work UW1 modifies a row. Unit of work UW2 reads that row before UW1 commits. If UW1 subsequently rolls the changes back, UW2 has read nonexisting data.
6. Under UR or CS, if the cursor is not updatable, the current row can be updated or deleted by other application processes in some cases. For example, buffering might cause the current row at the client to be different from the current row at the server. Moreover, when using currently committed semantics under CS, a row that is being read might have uncommitted updates pending. In this case, the currently committed version of the row is always returned to the application.

Summary of isolation levels

Table 2 lists the concurrency issues that are associated with different isolation levels.

Table 2. Summary of isolation levels

<table>
<thead>
<tr>
<th>Isolation level</th>
<th>Access to uncommitted data</th>
<th>Non-repeatable reads</th>
<th>Phantom reads</th>
</tr>
</thead>
<tbody>
<tr>
<td>Repeatable read (RR)</td>
<td>Not possible</td>
<td>Not possible</td>
<td>Not possible</td>
</tr>
<tr>
<td>Read stability (RS)</td>
<td>Not possible</td>
<td>Not possible</td>
<td>Possible</td>
</tr>
<tr>
<td>Cursor stability (CS)</td>
<td>Not possible</td>
<td>Possible</td>
<td>Possible</td>
</tr>
<tr>
<td>Uncommitted read (UR)</td>
<td>Possible</td>
<td>Possible</td>
<td>Possible</td>
</tr>
</tbody>
</table>

The isolation level affects not only the degree of isolation among applications but also the performance characteristics of an individual application, because the processing and memory resources that are required to obtain and free locks vary with the isolation level. The potential for deadlocks also varies with the isolation level. [Table 3 on page 26] provides a simple heuristic to help you choose an initial
Isolation levels

isolation level for your application.

Table 3. Guidelines for choosing an isolation level

<table>
<thead>
<tr>
<th>Application type</th>
<th>High data stability required</th>
<th>High data stability not required</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read-write transactions</td>
<td>RS</td>
<td>CS</td>
</tr>
<tr>
<td>Read-only transactions</td>
<td>RR or RS</td>
<td>UR</td>
</tr>
</tbody>
</table>

Table spaces

A table space is a storage structure containing tables, indexes, large objects, and long data. They are used to organize data in a database into logical storage groupings that relate to where data is stored on a system. Table spaces are stored in database partition groups.

Using table spaces to organize storage offers a number of benefits:

Recoverability
Putting objects that must be backed up or restored together into the same table space makes backup and restore operations more convenient, since you can backup or restore all the objects in table spaces with a single command. If you have partitioned tables and indexes that are distributed across table spaces, you can backup or restore only the data and index partitions that reside in a given table space.

More tables
There are limits to the number of tables that can be stored in any one table space; if you have a need for more tables than can be contained in a table space, you need only to create additional table spaces for them.

Storage flexibility
With DMS and SMS table spaces, you can specify which storage devices are used to store data. You could choose, for example, choose to store current, operational data in table spaces that reside on faster devices, and historical data in table spaces that reside on slower (and less expensive) devices.

Ability to isolate data in buffer pools for improved performance or memory utilization
If you have a set of objects (for example, tables, indexes) that are queried frequently, you can assign the table space in which they reside a buffer pool with a single CREATE or ALTER TABLESPACE statement. You can assign temporary table spaces to their own buffer pool to increase the performance of activities such as sorts or joins. In some cases, it might make sense to define smaller buffer pools for seldom-accessed data, or for applications that require very random access into a very large table; in such cases, data need not be kept in the buffer pool for longer than a single query.

Table spaces consist of one or more containers. A container can be a directory name, a device name, or a file name. A single table space can have several containers. It is possible for multiple containers (from one or more table spaces) to be created on the same physical storage device (although you will get the best performance if each container you create uses a different storage device). If you are using automatic storage table spaces, the creation and management of containers is handled automatically by the database manager. If you are not using automatic storage table spaces, you must define and manage containers yourself.
Figure 6 illustrates the relationship between tables and table spaces within a database, and the containers associated with that database.

The EMPLOYEE and DEPARTMENT tables are in the HUMANRES table space, which spans containers 0, 1, 2 and 3. The PROJECT table is in the SCHED table space in container 4. This example shows each container existing on a separate disk.

The database manager attempts to balance the data load across containers. As a result, all containers are used to store data. The number of pages that the database manager writes to a container before using a different container is called the **extent size**. The database manager does not always start storing table data in the first container.

Figure 6 on page 28 shows the HUMANRES table space with an extent size of two 4 KB pages, and four containers, each with a small number of allocated extents. The DEPARTMENT and EMPLOYEE tables both have seven pages, and span all four containers.
Character conversion

A string is a sequence of bytes that may represent characters. All the characters within a string have a common coding representation. In some cases, it may be necessary to convert these characters to a different coding representation, a process known as character conversion.

When character conversion is required, it is automatic. Applications do not need to explicitly invoke character conversion, because the DB2 database server and client perform all necessary character conversion automatically.

Character conversion can occur when an SQL statement is executed remotely. Consider, for example, the following scenarios in which the coding representations may be different at the sending and receiving systems:

- The values of host variables are sent from the application requester to the application server.
- The values of result columns are sent from the application server to the application requester.

Following is a list of terms used when discussing character conversion:

**character set**
A defined set of characters. For example, the following character set appears in several code pages:
- 26 non-accented letters A through Z
- 26 non-accented letters a through z
- digits 0 through 9
- , :, ; ? ( ) ” / - _ & + % * = < >

**code page**
A set of assignments of characters to code points. In the ASCII encoding scheme for code page 850, for example, “A” is assigned code point X’41’, and “B” is assigned code point X’42’. Within a code page, each code point has only one specific meaning. A code page is an attribute of the database. When an application program connects to the database, the database manager determines the code page of the application.
**Character conversion**

**code point**
A unique bit pattern that represents a character.

**encoding scheme**
A set of rules used to represent character data, for example:
- Single-Byte ASCII
- Single-Byte EBCDIC
- Double-Byte ASCII
- Mixed single- and double-byte ASCII

The following figure shows how a typical character set might map to different code points in two different code pages. Even with the same encoding scheme, there are many different code pages, and the same code point can represent a different character in different code pages. Furthermore, a byte in a character string does not necessarily represent a character from a single-byte character set (SBCS). Character strings are also used for mixed and bit data. *Mixed data* is a mixture of single-byte, double-byte, or multibyte characters. *Bit data* (columns defined as FOR BIT DATA, or BLOBs, or binary strings) is not associated with any character set.

The database manager determines code page attributes for all character strings when an application is bound to a database. The possible code page attributes are:

---

**Figure 8. Mapping a Character Set in Different Code Pages**

<table>
<thead>
<tr>
<th>code page: pp1 (ASCII)</th>
<th>code page: pp2 (EBCDIC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5</td>
<td>0 1 2 3 4 5</td>
</tr>
<tr>
<td>0  0  @  P</td>
<td>0  #  0</td>
</tr>
<tr>
<td>1  1  A  Q</td>
<td>1  $  A  J  1</td>
</tr>
<tr>
<td>2  *  2  B  R</td>
<td>2  s  %  B  K  S  2</td>
</tr>
<tr>
<td>3  3  C  S</td>
<td>3  t  C  L  T  3</td>
</tr>
<tr>
<td>4  4  D  T</td>
<td>4  u  *  D  M  U  4</td>
</tr>
<tr>
<td>5  %  5  E  U</td>
<td>5  v  (  E  N  V  5</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>E  &gt;  N</td>
<td>5/8  Ö</td>
</tr>
<tr>
<td>F  /  *  0</td>
<td>F  Å  ç  ;  Å  {</td>
</tr>
</tbody>
</table>

**code point**: 2F  **character set** ss1 (in code page pp1)  **character set** ss1 (in code page pp2)
Character conversion

Database code page
The database code page is stored in the database configuration file. The value is specified when the database is created and cannot be altered.

Application code page
The code page under which the application runs. This is not necessarily the same code page under which the application was bound.

Section code page
The code page under which the SQL statement runs. Typically, the section code page is the database code page. However, the Unicode code page (UTF-8) is used as the section code page if:

- The statement references a table that is created with the Unicode encoding scheme in a non-Unicode database
- The statement references a table function that is defined with PARAMETER CCSID UNICODE in a non-Unicode database

Code Page 0
This represents a string that is derived from an expression that contains a FOR BIT DATA value or a BLOB value.

Character string code pages have the following attributes:

- Columns can be in the database code page, the Unicode code page (UTF-8), or code page 0 (if defined as FOR BIT DATA or BLOB).
- Constants and special registers (for example, USER, CURRENT SERVER) are in the section code page. Constants are converted, if necessary, from the application code page to the database code page, and then to the section code page when an SQL statement is bound to the database.
- Input host variables are in the application code page. As of Version 8, string data in input host variables is converted, if necessary, from the application code page to the section code page before being used. The exception occurs when a host variable is used in a context where it is to be interpreted as bit data; for example, when the host variable is to be assigned to a column that is defined as FOR BIT DATA.

A set of rules is used to determine code page attributes for operations that combine string objects, such as scalar operations, set operations, or concatenation. Code page attributes are used to determine requirements for code page conversion of strings at run time.

Multicultural support and SQL statements
The coding of SQL statements is not language dependent. The SQL keywords must be typed as shown, although they may be typed in uppercase, lowercase, or mixed case. The names of database objects, host variables and program labels that occur in an SQL statement must be characters supported by your application code page.

The server does not convert file names. To code a file name, either use the ASCII invariant set, or provide the path in the hexadecimal values that are physically stored in the file system.

In a multibyte environment, there are four characters which are considered special that do not belong to the invariant character set. These characters are:

- The double-byte percentage and double-byte underscore characters used in LIKE processing.
Multicultural support and SQL statements

- The double-byte space character used for blank padding in graphic strings and in other places.
- The double-byte substitution character, used as a replacement during code page conversion when no mapping exists between a source code page and a target code page.

The code points for each of these characters, by code page, is as follows:

<table>
<thead>
<tr>
<th>Code Page</th>
<th>Double-Byte Percentage</th>
<th>Double-Byte Underscore</th>
<th>Double-Byte Space</th>
<th>Double-Byte Substitution Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>932</td>
<td>X’8193’</td>
<td>X’8151’</td>
<td>X’8140’</td>
<td>X’FCFC’</td>
</tr>
<tr>
<td>938</td>
<td>X’8193’</td>
<td>X’8151’</td>
<td>X’8140’</td>
<td>X’FCFC’</td>
</tr>
<tr>
<td>942</td>
<td>X’8193’</td>
<td>X’8151’</td>
<td>X’8140’</td>
<td>X’FCFC’</td>
</tr>
<tr>
<td>943</td>
<td>X’8193’</td>
<td>X’8151’</td>
<td>X’8140’</td>
<td>X’FCFC’</td>
</tr>
<tr>
<td>948</td>
<td>X’8193’</td>
<td>X’8151’</td>
<td>X’8140’</td>
<td>X’FCFC’</td>
</tr>
<tr>
<td>949</td>
<td>X’A3A5’</td>
<td>X’A3DF’</td>
<td>X’A1A1’</td>
<td>X’AFFE’</td>
</tr>
<tr>
<td>950</td>
<td>X’A248’</td>
<td>X’A1C4’</td>
<td>X’A140’</td>
<td>X’C8FE’</td>
</tr>
<tr>
<td>954</td>
<td>X’A1F3’</td>
<td>X’A1B2’</td>
<td>X’A1A1’</td>
<td>X’F4FE’</td>
</tr>
<tr>
<td>964</td>
<td>X’A2E8’</td>
<td>X’A2A5’</td>
<td>X’A1A1’</td>
<td>X’DFE’</td>
</tr>
<tr>
<td>970</td>
<td>X’A3A5’</td>
<td>X’A3DF’</td>
<td>X’A1A1’</td>
<td>X’AFFE’</td>
</tr>
<tr>
<td>1381</td>
<td>X’A3A5’</td>
<td>X’A3DF’</td>
<td>X’A1A1’</td>
<td>X’FEFE’</td>
</tr>
<tr>
<td>1383</td>
<td>X’A3A5’</td>
<td>X’A3DF’</td>
<td>X’A1A1’</td>
<td>X’A1A1’</td>
</tr>
<tr>
<td>1388</td>
<td>X’FF05’</td>
<td>X’FF3F’</td>
<td>X’3000’</td>
<td>X’FFFD’</td>
</tr>
<tr>
<td>1363</td>
<td>X’A3A5’</td>
<td>X’A3DF’</td>
<td>X’A1A1’</td>
<td>X’A1E0’</td>
</tr>
<tr>
<td>1386</td>
<td>X’A3A5’</td>
<td>X’A3DF’</td>
<td>X’A1A1’</td>
<td>X’FEFE’</td>
</tr>
<tr>
<td>5039</td>
<td>X’8193’</td>
<td>X’8151’</td>
<td>X’8140’</td>
<td>X’FCFC’</td>
</tr>
<tr>
<td>1392</td>
<td>X’A3A5’</td>
<td>X’A3DF’</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Code Page 5488 is equivalent to 1392 for the EXPORT, IMPORT, and LOAD utilities.

For Unicode databases, the GRAPHIC space is X’0020’, which is different from the GRAPHIC space of X’3000’ used for eucJP (Extended UNIX Code - Japan) and eucTW (Extended UNIX Code - Taiwan) databases.

Both X’0020’ and X’3000’ are space characters in the Unicode standard. The difference in the GRAPHIC space code points should be taken into consideration when comparing data from these EUC databases to data from a Unicode database.

Connecting to distributed relational databases

Distributed relational databases are built on formal requester-server protocols and functions.

An application requester supports the application end of a connection. It transforms a database request from the application into communication protocols suitable for use in the distributed database network. These requests are received and processed
Connecting to distributed relational databases

by a database server at the other end of the connection. Working together, the application requester and the database server handle communication and location considerations, so that the application can operate as if it were accessing a local database.

An application process must connect to a database manager’s application server before SQL statements that reference tables or views can be executed. The CONNECT statement establishes a connection between an application process and its server.

There are two types of CONNECT statements:
- CONNECT (Type 1) supports the single database per unit of work (Remote Unit of Work) semantics.
- CONNECT (Type 2) supports the multiple databases per unit of work (Application-Directed Distributed Unit of Work) semantics.

The DB2 call level interface (CLI) and embedded SQL support a connection mode called concurrent transactions, which allows multiple connections, each of which is an independent transaction. An application can have multiple concurrent connections to the same database.

The application server can be local to or remote from the environment in which the process is initiated. An application server is present, even if the environment is not using distributed relational databases. This environment includes a local directory that describes the application servers that can be identified in a CONNECT statement.

The application server runs the bound form of a static SQL statement that references tables or views. The bound statement is taken from a package that the database manager has previously created through a bind operation.

For the most part, an application connected to an application server can use statements and clauses that are supported by the application server’s database manager. This is true even if an application is running through the application requester of a database manager that does not support some of those statements and clauses.

Event monitors that write to tables, files, and pipes

Some event monitors can be configured to write information about database events to tables, pipes, or files.

Event monitors are used to collect information about the database and any connected applications when specified events occur. Events represent transitions in database activity such as connections, deadlocks, statements, or transactions. You can define an event monitor by the type of event or events you want it to monitor. For example, a deadlock event monitor waits for a deadlock to occur; when one does, it collects information about the applications involved and the locks in contention.

To create an event monitor, use the CREATE EVENT MONITOR SQL statement. Event monitors collect event data only when they are active. To activate or deactivate an event monitor, use the SET EVENT MONITOR STATE SQL statement. The status of an event monitor (whether it is active or inactive) can be determined by the SQL function EVENT_MON_STATE.
Event monitors that write to tables, files, and pipes

When the CREATE EVENT MONITOR SQL statement is executed, the definition of the event monitor it creates is stored in the following database system catalog tables:

- SYSCAT.EVENTMONITORS: event monitors defined for the database.
- SYSCAT.EVENTS: events monitored for the database.
- SYSCAT.EVENTTABLES: target tables for table event monitors.

Each event monitor has its own private logical view of the instance's data in the monitor elements. If a particular event monitor is deactivated and then reactivated, its view of these counters is reset. Only the newly activated event monitor is affected; all other event monitors will continue to use their view of the counter values (plus any new additions).

Event monitor output can be directed to non-partitioned SQL tables, a file, or a named pipe.

Note: The deprecated detailed deadlock event monitor, DB2DETAILDEADLOCK, is created by default for each database and starts when the database is activated. Avoid the overhead this event monitor incurs by dropping it. The use of the DB2DETAILDEADLOCK monitor element is no longer recommended. This deprecated event monitor might be removed in a future release. Use the CREATE EVENT MONITOR FOR LOCKING statement to monitor lock-related events, such as lock timeouts, lock waits, and deadlocks.

Database partitioning across multiple database partitions

The database manager allows great flexibility in spreading data across multiple database partitions (nodes) of a partitioned database. Users can choose how to distribute their data by declaring distribution keys, and can determine which and how many database partitions their table data can be spread across by selecting the database partition group and table space in which the data should be stored.

In addition, a distribution map (which is updatable) specifies the mapping of distribution key values to database partitions. This makes it possible for flexible workload parallelization across a partitioned database for large tables, while allowing smaller tables to be stored on one or a small number of database partitions if the application designer so chooses. Each local database partition may have local indexes on the data it stores to provide high performance local data access.

In a partitioned database, the distribution key is used to distribute table data across a set of database partitions. Index data is also partitioned with its corresponding tables, and stored locally at each database partition.

Before database partitions can be used to store data, they must be defined to the database manager. Database partitions are defined in a file called db2nodes.cfg.

The distribution key for a table in a table space on a partitioned database partition group is specified in the CREATE TABLE statement or the ALTER TABLE statement. If not specified, a distribution key for a table is created by default from the first column of the primary key. If no primary key is defined, the default distribution key is the first column defined in that table that has a data type other than a long or a LOB data type. Tables in partitioned databases must have at least
Database partitioning across multiple database partitions

one column that is neither a long nor a LOB data type. A table in a table space that
is in a single partition database partition group will have a distribution key only if
it is explicitly specified.

Rows are placed in a database partition as follows:
1. A hashing algorithm (database partitioning function) is applied to all of the
   columns of the distribution key, which results in the generation of a
distribution map index value.
2. The database partition number at that index value in the distribution map
   identifies the database partition in which the row is to be stored.

The database manager supports partial declustering, which means that a table can be
distributed across a subset of database partitions in the system (that is, a database
partition group). Tables do not have to be distributed across all of the database
partitions in the system.

The database manager has the capability of recognizing when data being accessed
for a join or a subquery is located at the same database partition in the same
database partition group. This is known as table collocation. Rows in collocated
tables with the same distribution key values are located on the same database
partition. The database manager can choose to perform join or subquery processing
at the database partition in which the data is stored. This can have significant
performance advantages.

Collocated tables must:
- Be in the same database partition group, one that is not being redistributed.
  (During redistribution, tables in the database partition group may be using
different distribution maps – they are not collocated.)
- Have distribution keys with the same number of columns.
- Have the corresponding columns of the distribution key be database
  partition-compatible.
- Be in a single partition database partition group defined on the same database
  partition.

Large object behavior in partitioned tables

A partitioned table uses a data organization scheme in which table data is divided
across multiple storage objects, called data partitions or ranges, according to values
in one or more table partitioning key columns of the table. Data from a given table
is partitioned into multiple storage objects based on the specifications provided in
the PARTITION BY clause of the CREATE TABLE statement. These storage objects
can be in different table spaces, in the same table space, or a combination of both.

A large object for a partitioned table is, by default, stored in the same table space
as its corresponding data object. This applies to partitioned tables that use only one
table space or use multiple table spaces. When a partitioned table’s data is stored
in multiple table spaces, the large object data is also stored in multiple table
spaces.

Use the LONG IN clause of the CREATE TABLE statement to override this default
behavior. You can specify a list of table spaces for the table where long data is to
be stored. If you choose to override the default behavior, the table space specified
in the LONG IN clause must be a large table space. If you specify that long data
be stored in a separate table space for one or more data partitions, you must do so
for all the data partitions of the table. That is, you cannot have long data stored remotely for some data partitions and stored locally for others. Whether you are using the default behavior or the LONG IN clause to override the default behavior, a long object is created to correspond to each data partition. For SMS table spaces, the long data must reside in the same table space as the data object it belongs to. All the table spaces used to store long data objects corresponding to each data partition must have the same: pagesize, extentsize, storage mechanism (DMS or SMS), and type (regular or large). Remote large table spaces must be of type LARGE and cannot be SMS.

For example, the following CREATE TABLE statement creates objects for the CLOB data for each data partition in the same table space as the data:

```
CREATE TABLE document(id INT, contents CLOB)
PARTITION BY RANGE(id)
(STARTING FROM 1 ENDING AT 100 IN tbsp1,
 STARTING FROM 101 ENDING AT 200 IN tbsp2,
 STARTING FROM 201 ENDING AT 300 IN tbsp3,
 STARTING FROM 301 ENDING AT 400 IN tbsp4);
```

You can use LONG IN to place the CLOB data in one or more large table spaces, distinct from those the data is in.

```
CREATE TABLE document(id INT, contents CLOB)
PARTITION BY RANGE(id)
(STARTING FROM 1 ENDING AT 100 IN tbsp1 LONG IN largel,
 STARTING FROM 101 ENDING AT 200 IN tbsp2 LONG IN largel,
 STARTING FROM 201 ENDING AT 300 IN tbsp3 LONG IN large2,
 STARTING FROM 301 ENDING AT 400 IN tbsp4 LONG IN large2);
```

Note: Only a single LONG IN clause is allowed at the table level and for each data partition.

### DB2 federated systems

#### Federated systems

A federated system is a special type of distributed database management system (DBMS). A federated system consists of a DB2 instance that operates as a federated server, a database that acts as the federated database, one or more data sources, and clients (users and applications) that access the database and data sources.

With a federated system, you can send distributed requests to multiple data sources within a single SQL statement. For example, you can join data that is located in a DB2 table, an Oracle table, and an XML tagged file in a single SQL statement. The following figure shows the components of a federated system and a sample of the data sources you can access.
The power of a federated system is in its ability to:

- Correlate data from local tables and remote data sources, as if all the data is stored locally in the federated database
- Update data in relational data sources, as if the data is stored in the federated database
- Move data to and from relational data sources
- Take advantage of the data source processing strengths, by sending requests to the data sources for processing
- Compensate for SQL limitations at the data source by processing parts of a distributed request at the federated server

**What is a data source?**

In a federated system, a *data source* can be a relational database (such as Oracle or Sybase) or a nonrelational data source (such as an XML tagged file).

Through some data sources you can access other data sources. For example, with the ODBC wrapper you can access IBM InfoSphere™ Classic Federation Server for z/OS data sources such as DB2 for z/OS, IMS™, CA-IDMS, CA-Datacom, Software AG Adabas, and VSAM.

The method, or protocol, used to access a data source depends on the type of data source. For example, DRDA is used to access DB2 for z/OS data sources.
Data sources are autonomous. For example, the federated server can send queries to Oracle data sources at the same time that Oracle applications can access these data sources. A federated system does not monopolize or restrict access to the other data sources, beyond integrity and locking constraints.

**The federated database**

To end users and client applications, data sources appear as a single collective database in the DB2 database system. Users and applications interface with the *federated database* that is managed by the federated server.

The federated database contains a system catalog that stores information about data. The federated database system catalog contains entries that identify data sources and their characteristics. The federated server consults the information stored in the federated database system catalog and the data source wrapper to determine the best plan for processing SQL statements.

The federated system processes SQL statements as if the data from the data sources were ordinary relational tables or views within the federated database. As a result:

- The federated system can correlate relational data with data in nonrelational formats. This is true even when the data sources use different SQL dialects, or do not support SQL at all.
- The characteristics of the federated database take precedence when there are differences between the characteristics of the federated database and the characteristics of the data sources. Query results conform to DB2 semantics, even if data from other non-DB2 data sources is used to compute the query result.

Examples:

- The code page that the federated server uses is different than the code page used that the data source uses. In this case, character data from the data source is converted based on the code page used by the federated database, when that data is returned to a federated user.
- The collating sequence that the federated server uses is different than the collating sequence that the data source uses. In this case, any sort operations on character data are performed at the federated server instead of at the data source.

**The SQL compiler**

The DB2 SQL compiler gathers information to help it process queries.

To obtain data from data sources, users and applications submit queries in SQL to the federated database. When a query is submitted, the DB2 SQL compiler consults information in the global catalog and the data source wrapper to help it process the query. This includes information about connecting to the data source, server information, mappings, index information, and processing statistics.

**Wrappers and wrapper modules**

*Wrappers* are mechanisms by which the federated database interacts with data sources. The federated database uses routines stored in a library called a *wrapper module* to implement a wrapper.

These routines allow the federated database to perform operations such as connecting to a data source and retrieving data from it iteratively. Typically, the...
Wrappers and wrapper modules

federated instance owner uses the CREATE WRAPPER statement to register a wrapper in the federated database. You can register a wrapper as fenced or trusted using the DB2_FENCED wrapper option.

You create one wrapper for each type of data source that you want to access. For example, you want to access three DB2 for z/OS database tables, one DB2 for System i table, two Informix tables, and one Informix view. In this case, you need to create one wrapper for the DB2 data source objects and one wrapper for the Informix data source objects. After these wrappers are registered in the federated database, you can use these wrappers to access other objects from those data sources. For example, you can use the DRDA wrapper with all DB2 family data source objects—DB2 Database for Linux, UNIX, and Windows, DB2 for z/OS, DB2 for System i, and DB2 Server for VM and VSE.

You use the server definitions and nicknames to identify the specifics (name, location, and so forth) of each data source object.

A wrapper performs many tasks. Some of these tasks are:

• It connects to the data source. The wrapper uses the standard connection API of the data source.
• It submits queries to the data source.
  – For data sources that support SQL, the query is submitted in SQL.
  – For data sources that do not support SQL, the query is translated into the native query language of the source or into a series of source API calls.
• It receives results sets from the data source. The wrapper uses the data source standard APIs for receiving results set.
• It responds to federated database queries about the default data type mappings for a data source. The wrapper contains the default type mappings that are used when nicknames are created for a data source object. For relational wrappers, data type mappings that you create override the default data type mappings. User-defined data type mappings are stored in the global catalog.
• It responds to federated database queries about the default function mappings for a data source. The federated database needs data type mapping information for query planning purposes. The wrapper contains information that the federated database needs to determine if DB2 functions are mapped to functions of the data source, and how the functions are mapped. This information is used by the SQL Compiler to determine if the data source is able to perform the query operations. For relational wrappers, function mappings that you create override the default function type mappings. User-defined function mappings are stored in the global catalog.

Wrapper options are used to configure the wrapper or to define how IBM InfoSphere Federation Server uses the wrapper.

Server definitions and server options

After wrappers are created for the data sources, the federated instance owner defines the data sources to the federated database.

The instance owner supplies a name to identify the data source, and other information that pertains to the data source. This information includes:

• The type and version of the data source
• The database name for the data source (RDBMS only)
• Metadata that is specific to the data source
Server definitions and server options

For example, a DB2 family data source can have multiple databases. The definition must specify which database the federated server can connect to. In contrast, an Oracle data source has one database, and the federated server can connect to the database without knowing its name. The database name is not included in the federated server definition of an Oracle data source.

The name and other information that the instance owner supplies to the federated server are collectively called a server definition. Data sources answer requests for data and are servers in their own right.

The CREATE SERVER and ALTER SERVER statements are used to create and modify a server definition.

Some of the information within a server definition is stored as server options. When you create server definitions, it is important to understand the options that you can specify about the server.

Server options can be set to persist over successive connections to the data source, or set for the duration of a single connection.

User mappings

A user mapping is an association between an authorization ID on the federated server and the information that is required to connect to the remote data source.

To create a user mapping, you use the CREATE USER MAPPING statement. In the statement, you specify the local authorization ID, the local name of the remote data source server as specified in the server definition, and the remote ID and password.

For example, assume that you created a server definition for a remote server and specified 'argon' as the local name for the remote server. To give Mary access to the remote server, create this user mapping:

```
CREATE USER MAPPING FOR Mary
SERVER argon
OPTIONS (REMOTE_AUTHID 'remote_ID', REMOTE_PASSWORD 'remote_pw')
```

When Mary issues an SQL statement to connect to the remote server, the federated server performs these steps:
1. Retrieves Mary’s user mapping
2. Decrypts the remote password 'remote_pw' that is associated with the remote server
3. Calls the wrapper to connect to the remote server
4. Passes the remote ID 'remote_ID' and the decrypted remote password to the wrapper
5. Creates a connection to the remote server for Mary

By default, the federated server stores user mapping in the SYSCAT.USEROPTIONS view in the global catalog and encrypts the remote passwords. As an alternative, you can use an external repository, for example a file or an LDAP server, to store user mappings. To provide the interface between the federated server and the external repository, you create a user mapping plug-in.
User mappings

No matter how you store user mappings, carefully restrict access to them. If user mappings are compromised, data in the remote databases might be vulnerable to unauthorized activity.

Nicknames and data source objects

A nickname is an identifier that you use to identify the data source object that you want to access. The object that a nickname identifies is referred to as a data source object.

A nickname is not an alternative name for a data source object in the same way that an alias is an alternative name. A nickname is the pointer by which the federated server references the object. Nicknames are typically defined with the CREATE NICKNAME statement along with specific nickname column options and nickname options.

When a client application or a user submits a distributed request to the federated server, the request does not need to specify the data sources. Instead, the request references the data source objects by their nicknames. The nicknames are mapped to specific objects at the data source. These mappings eliminate the need to qualify the nicknames by data source names. The location of the data source objects is transparent to the client application or the user.

Suppose that you define the nickname `DEPT` to represent an Informix database table called `NFX1.PERSON`. The statement `SELECT * FROM DEPT` is allowed from the federated server. However, the statement `SELECT * FROM NFX1.PERSON` is not allowed from the federated server (except in a pass-through session) unless there is a local table on the federated server named `NFX1.PERSON`.

When you create a nickname for a data source object, metadata about the object is added to the global catalog. The query optimizer uses this metadata, as well as information in the wrapper, to facilitate access to the data source object. For example, if a nickname is for a table that has an index, the global catalog contains information about the index, and the wrapper contains the mappings between the DB2 data types and the data source data types.

Nicknames for objects that use label-based access control (LBAC) are not cached. Therefore, data in the object remains secure. For example, if you use the Oracle (Net8) wrapper to create a nickname on a table that uses Oracle Label Security, the table is automatically identified as secure. The resulting nickname data cannot be cached. As a result, materialized query tables cannot be created on it. Using LBAC ensures that the information is viewed only by users who have the appropriate security privileges. For nicknames that were created before LBAC was supported, you must use the ALTER NICKNAME statement to disallow caching. LBAC is supported by both the DRDA (for data sources that use DB2 for Linux, UNIX, and Windows version 9.1 and later) and the Net8 wrapper.

Nickname column options

You can supply the global catalog with additional metadata information about the nicknamed object. This metadata describes values in certain columns of the data source object. You assign this metadata to parameters that are called nickname column options.

The nickname column options tell the wrapper to handle the data in a column differently than it normally would handle it. The SQL compiler and query optimizer use the metadata to develop better plans for accessing the data.
Nickname column options

Nickname column options are used to provide other information to the wrapper as well. For example for XML data sources, a nickname column option is used to tell the wrapper the XPath expression to use when the wrapper parses the column out of the XML document.

With federation, the DB2 server treats the data source object that a nickname references as if it is a local DB2 table. As a result, you can set nickname column options for any data source object that you create a nickname for. Some nickname column options are designed for specific types of data sources and can be applied only to those data sources.

Suppose that a data source has a collating sequence that differs from the federated database collating sequence. The federated server typically would not sort any columns containing character data at the data source. It would return the data to the federated database and perform the sort locally. However, suppose that the column is a character data type (CHAR or VARCHAR) and contains only numeric characters ('0','1',...,'9'). You can indicate this by assigning a value of 'Y' to the NUMERIC_STRING nickname column option. This gives the DB2 query optimizer the option of performing the sort at the data source. If the sort is performed remotely, you can avoid the overhead of porting the data to the federated server and performing the sort locally.

You can define nickname column options for relational nicknames using the ALTER NICKNAME statements. You can define nickname column options for nonrelational nicknames using the CREATE NICKNAME and ALTER NICKNAME statements.

Data type mappings

The data types at the data source must map to corresponding DB2 data types so that the federated server can retrieve data from data sources.

Some examples of default data type mappings are:
- The Oracle type FLOAT maps to the DB2 type DOUBLE
- The Oracle type DATE maps to the DB2 type TIMESTAMP
- The DB2 for z/OS™ type DATE maps to the DB2 type DATE

For most data sources, the default type mappings are in the wrappers. The default type mappings for DB2 data sources are in the DRDA wrapper. The default type mappings for Informix are in the INFORMIX wrapper, and so forth.

For some nonrelational data sources, you must specify data type information in the CREATE NICKNAME statement. The corresponding DB2 data types must be specified for each column of the data source object when the nickname is created. Each column must be mapped to a particular field or column in the data source object.

For relational data sources, you can override the default data type mappings. For example, by default the Informix INTEGER data type maps to the DB2 INTEGER data type. You could override the default mappings and map Informix's INTEGER data type to DB2 DECIMAL(10,0) data type.
The federated server

The federated server
The DB2 server in a federated system is referred to as the federated server. Any number of DB2 instances can be configured to function as federated servers. You can use existing DB2 instances as your federated servers, or you can create new ones specifically for the federated system.

The DB2 instance that manages the federated system is called a server because it responds to requests from end users and client applications. The federated server often sends parts of the requests it receives to the data sources for processing. A pushdown operation is an operation that is processed remotely. The DB2 instance that manages the federated system is referred to as the federated server, even though it acts as a client when it pushes down requests to the data sources.

Like any other application server, the federated server is a database manager instance. Application processes connect and submit requests to the database within the federated server. However, two main features distinguish it from other application servers:

- A federated server is configured to receive requests that might be partially or entirely intended for data sources. The federated server distributes these requests to the data sources.
- Like other application servers, a federated server uses DRDA communication protocols (over TCP/IP) to communicate with DB2 family instances. However, unlike other application servers, a federated server uses the native client of the data source to access the data source. For example, a federated server uses the Sybase Open Client to access Sybase data sources and an Microsoft SQL Server ODBC Driver to access Microsoft SQL Server data sources.
Supported data sources

There are many data sources that you can access using a federated system.

IBM InfoSphere Federation Server supports the data sources shown in the following tables. The first table lists the requirements for data client software. The client software must be acquired separately unless specified otherwise.

You must install the client software for the data sources that you want to access. The client software must be installed on the same system as IBM InfoSphere Federation Server. You also need the appropriate Java SDK to use some tools such as the DB2 Control Center and to create and run Java applications, including stored procedures and user-defined functions.

For the most up-to-date information, see the [Data source requirements page](#) on the Web.

*Table 5. Supported data sources, client software requirements, and support from 32-bit operating systems.*

<table>
<thead>
<tr>
<th>Data source</th>
<th>Supported versions</th>
<th>Client software</th>
<th>32-bit hardware architecture and operating system</th>
<th>Windows</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>X86-32</td>
<td></td>
</tr>
<tr>
<td>BioRS</td>
<td>5.2, 5.3</td>
<td>None</td>
<td>Linux, RedHat Enterprise Linux (RHEL), SUSE</td>
<td>Y</td>
</tr>
<tr>
<td>DB2 for Linux, UNIX, and Windows</td>
<td>8.1.x, 8.2.x, 9.1, 9.5, 9.7</td>
<td>None</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>DB2 for z/OS</td>
<td>7.x, 8.x, 9.x</td>
<td>DB2 Connect V9.7</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>DB2 for System i</td>
<td>5.2, 5.3, 5.4, 6.1</td>
<td>DB2 Connect V9.7</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>DB2 Server for VSE and VM</td>
<td>7.2, 7.4</td>
<td>DB2 Connect V9.7</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Flat files</td>
<td>None</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Informix</td>
<td>Informix XPS 8.50, 8.51 and Informix IDS IDS 7.31, IDS 9.40, IDS 10.0, 11.5, 11.10</td>
<td>Informix Client SDK version 2.81.TC2 or later, 3.0 required for SLES 10 on Power</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>JDBC</td>
<td>3.0 or later</td>
<td>JDBC drivers that comply with JDBC 3.0 or later</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Microsoft SQL Server</td>
<td>Microsoft SQL Server 2000/SP4, 2005, 2008</td>
<td>For Windows, Microsoft SQL Server Client ODBC 3.0 (or later) driver. For Unix, DataDirect ODBC 5.3</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>MQ</td>
<td>MQ7</td>
<td>MQ7</td>
<td>Y</td>
<td>Y</td>
</tr>
</tbody>
</table>
## Supported data sources

Table 5. Supported data sources, client software requirements, and support from 32-bit operating systems. (continued)

<table>
<thead>
<tr>
<th>Data source</th>
<th>Supported versions</th>
<th>Client software</th>
<th>32-bit hardware architecture and operating system</th>
</tr>
</thead>
<tbody>
<tr>
<td>ODBC</td>
<td>3.0</td>
<td>ODBC drivers that comply with ODBC 3.0, or later**</td>
<td>Y</td>
</tr>
<tr>
<td>OLE DB</td>
<td>2.7, 2.8</td>
<td>OLE DB 2.0, or later</td>
<td>Y</td>
</tr>
<tr>
<td>Oracle</td>
<td>10g, 10gR2, 11g, 11gR1</td>
<td>Oracle NET client 10.0 - 10.1, 10.2.0.1 with patch 3807408, 10.1.0.3 with patch 3807408, 11.1.0.6.0</td>
<td>Y</td>
</tr>
<tr>
<td>Sybase</td>
<td>Sybase ASE 12.5, 15.0</td>
<td>Sybase Open Client 12.5 - 15.0</td>
<td>Y</td>
</tr>
<tr>
<td>Teradata</td>
<td>2.5, 2.6, 12</td>
<td>Shared Common Components for Internationalization for Teradata (tdicu) version 1.01 or later, Teradata Generic Security Services (TeraGSS) version 6.01 or later, and the software on the following operating systems: For Windows Teradata client TTU 8.0 or later and the Teradata API library CLIv2 4.8.0 or later For UNIX and Linux Teradata Call-Level Interface Version 2 CLIv2 Release 4.8.0 or later</td>
<td>Y</td>
</tr>
<tr>
<td>Web Services</td>
<td>WSDL 1.0, 1.1</td>
<td>None</td>
<td>Y</td>
</tr>
<tr>
<td>XML</td>
<td>XML1.0, XML1.1</td>
<td>None</td>
<td>Y</td>
</tr>
</tbody>
</table>

** ODBC can be used to access RedBrick 6.20cu5 and InfoSphere Classic Federation V8.2 and above, among other data sources.
Table 6. Support from 64-bit operating systems.

<table>
<thead>
<tr>
<th>64-bit hardware architecture</th>
<th>X86-64</th>
<th>X86-64</th>
<th>Power</th>
<th>Itanium®</th>
<th>Power</th>
<th>Sparc</th>
<th>zSeries®</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operating system</td>
<td>Linux</td>
<td>RHEL</td>
<td>SUSE</td>
<td>AIX®</td>
<td>HP-UX</td>
<td>Linux</td>
<td>RHEL SUSE</td>
</tr>
<tr>
<td>Data source</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BioRS</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>DB2 for Linux, UNIX, and Windows</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>DB2 for z/OS</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>DB2 for System i</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>DB2 Server for VSE and VM</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Informix</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>JDBC</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Microsoft Excel</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Microsoft SQL Server</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>MQ</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>ODBC</td>
<td>Y</td>
<td>Y</td>
<td>Y***</td>
<td>Y</td>
<td>Y***</td>
<td>Y***</td>
<td>Y</td>
</tr>
<tr>
<td>OLE DB</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Oracle</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Script</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Sybase</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Teradata</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Web Services</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>XML</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
</tbody>
</table>

*** ODBC can be used to access RedBrick 6.20cu5 and IBM InfoSphere Classic Federation Server for z/OS using 32-bit and 64-bit clients.

The federated database system catalog

The federated database system catalog contains information about the objects in the federated database and information about objects at the data sources.

The catalog in a federated database is called the global catalog because it contains information about the entire federated system. DB2 query optimizer uses the information in the global catalog and the data source wrapper to plan the best way to process SQL statements. The information stored in the global catalog includes remote and local information, such as column names, column data types, column default values, index information, and statistics information.

Remote catalog information is the information or name used by the data source. Local catalog information is the information or name used by the federated database. For example, suppose a remote table includes a column with the name of
The federated database system catalog

EMPNO. The global catalog would store the remote column name as EMPNO. Unless you designate a different name, the local column name will be stored as EMPNO. You can change the local column name to Employee_Number. Users submitting queries which include this column will use Employee_Number in their queries instead of EMPNO. You use the ALTER NICKNAME statement to change the local name of the data source columns.

For relational and nonrelational data sources, the information stored in the global catalog includes both remote and local information.

To see the data source table information that is stored in the global catalog, query the SYSCAT.TABLES, SYSCAT.NICKNAMES, SYSCAT.TABOPTIONS, SYSCAT.INDEXES, SYSCAT.INDEXOPTIONS, SYSCAT.COLUMNS, and SYSCAT.COLOPTIONS catalog views in the federated database.

The global catalog also includes other information about the data sources. For example, the global catalog includes information that the federated server uses to connect to the data source and map the federated user authorizations to the data source user authorizations. The global catalog contains attributes about the data source that you explicitly set, such as server options.

The query optimizer

As part of the SQL compiler process, the query optimizer analyzes a query. The compiler develops alternative strategies, called access plans, for processing the query.

Access plans might call for the query to be:

- Processed by the data sources
- Processed by the federated server
- Processed partly by the data sources and partly by the federated server

The query optimizer evaluates the access plans primarily on the basis of information about the data source capabilities and the data. The wrapper and the global catalog contain this information. The query optimizer decomposes the query into segments that are called query fragments. Typically it is more efficient to pushdown a query fragment to a data source, if the data source can process the fragment. However, the query optimizer takes into account other factors such as:

- The amount of data that needs to be processed
- The processing speed of the data source
- The amount of data that the fragment will return
- The communication bandwidth
- Whether there is a usable materialized query table on the federated server that represents the same query result

The query optimizer generates access plan alternatives for processing a query fragment. The plan alternatives perform varying amounts of work locally on the federated server and on the remote data sources. Because the query optimizer is cost-based, it assigns resource consumption costs to the access plan alternatives. The query optimizer then chooses the plan that will process the query with the least resource consumption cost.

If any of the fragments are to be processed by data sources, the federated database submits these fragments to the data sources. After the data sources process the
The query optimizer

fragments, the results are retrieved and returned to the federated database. If the federated database performed any part of the processing, it combines its results with the results retrieved from the data source. The federated database then returns all results to the client.
Collating sequences

The order in which character data is sorted in a database depends on the structure of the data and the collating sequence defined for the database.

Suppose that the data in a database is all uppercase letters and does not contain any numeric or special characters. A sort of the data should result in the same output, regardless of whether the data is sorted at the data source or at the federated database. The collating sequence used by each database should not impact the sort results. Likewise, if the data in the database is all lowercase letters or all numeric characters, a sort of the data should produce the same results regardless of where the sort actually is performed.

If the data consists of any of the following structures:
- A combination of letters and numeric characters
- Both uppercase and lowercase letters
- Special characters such as @, #, €

Sorting this data can result in different outputs, if the federated database and the data source use different collating sequences.

In general terms, a collating sequence is a defined ordering for character data that determines whether a particular character sorts higher, lower, or the same as another character.

How collating sequences determine sort orders

A collating sequence determines the sort order of the characters in a coded character set.

A character set is the aggregate of characters that are used in a computer system or programming language. In a coded character set, each character is assigned to a different number within the range of 0 to 255 (or the hexadecimal equivalent thereof). The numbers are called code points; the assignments of numbers to characters in a set are collectively called a code page.

In addition to being assigned to a character, a code point can be mapped to the character’s position in a sort order. In technical terms, then, a collating sequence is the collective mapping of a character set’s code points to the sort order positions of the set’s characters. A character’s position is represented by a number; this number is called the weight of the character. In the simplest collating sequence, called an identity sequence, the weights are identical to the code points.

Example: Database ALPHA uses the default collating sequence of the EBCDIC code page. Database BETA uses the default collating sequence of the ASCII code page. Sort orders for character strings at these two databases would differ:

```
SELECT ....
    ORDER BY COL2
```

<table>
<thead>
<tr>
<th>EBCDIC-Based Sort</th>
<th>ASCII-Based Sort</th>
</tr>
</thead>
<tbody>
<tr>
<td>COL2</td>
<td>COL2</td>
</tr>
<tr>
<td>V1G</td>
<td>V1G</td>
</tr>
<tr>
<td>Y2W</td>
<td>Y2W</td>
</tr>
<tr>
<td>7AB</td>
<td>7AB</td>
</tr>
</tbody>
</table>

Example: Similarly, character comparisons in a database depend on the collating sequence defined for that database. Database ALPHA uses the default collating
How collating sequences determine sort orders

sequence of the EBCDIC code page. Database BETA uses the default collating sequence of the ASCII code page. Character comparisons at these two databases would yield different results:

```sql
SELECT.....
WHERE COL2 > 'TT3'
```

<table>
<thead>
<tr>
<th>EBCDIC-Based Results</th>
<th>ASCII-Based Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>COL2</td>
<td>COL2</td>
</tr>
<tr>
<td>TW4</td>
<td>TW4</td>
</tr>
<tr>
<td>X82</td>
<td>X82</td>
</tr>
<tr>
<td>39G</td>
<td></td>
</tr>
</tbody>
</table>

Setting the local collating sequence to optimize queries

Administrators can create federated databases with a particular collating sequence that matches a data source collating sequence.

Then for each data source server definition, the COLLATING_SEQUENCE server option is set to ‘Y’. This setting tells the federated database that the collating sequences of the federated database and the data source match.

You set the federated database collating sequence as part of the CREATE DATABASE command. Through this command, you can specify one of the following sequences:

- An identity sequence
- A system sequence (the sequence used by the operating system that supports the database)
- A customized sequence (a predefined sequence that the DB2 database system supplies or that you define yourself)

Suppose that the data source is DB2 for z/OS. Sorts that are defined in an ORDER BY clause are implemented by a collating sequence based on an EBCDIC code page. To retrieve DB2 for z/OS data sorted in accordance with ORDER BY clauses, configure the federated database so that it uses the predefined collating sequence based on the appropriate EBCDIC code page.
Setting the local collating sequence to optimize queries
Chapter 2. Language elements
The basic symbols of keywords and operators in the SQL language are single-byte characters that are part of all IBM character sets. Characters of the language are classified as letters, digits, or special characters.

A letter is any of the 26 uppercase (A through Z) or 26 lowercase (a through z) letters. Letters also include three code points reserved as alphabetic extenders for national languages (#, @, and $ in the United States). However these three code points should be avoided, especially for portable applications, because they represent different characters depending on the CCSID. Letters also include the alphabets from the extended character sets. Extended character sets contain additional alphabetic characters; for example, those with diacritical marks (á is an example of a diacritical mark). The available characters depend on the code page in use.

A digit is any of the characters 0 through 9.

A special character is any of the characters listed below:

<table>
<thead>
<tr>
<th>Character</th>
<th>Description</th>
<th>Character</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;</td>
<td>space or blank</td>
<td>-</td>
<td>minus sign</td>
</tr>
<tr>
<td>&quot;</td>
<td>quotation mark or</td>
<td>.</td>
<td>period</td>
</tr>
<tr>
<td>&quot;</td>
<td>double quote or</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;</td>
<td>double quotation mark</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%</td>
<td>percent</td>
<td>/</td>
<td>slash</td>
</tr>
<tr>
<td>&amp;</td>
<td>ampersand</td>
<td>:</td>
<td>colon</td>
</tr>
<tr>
<td>'</td>
<td>apostrophe or single</td>
<td>;</td>
<td>semicolon</td>
</tr>
<tr>
<td></td>
<td>quote or single</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>quotation mark</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(</td>
<td>left parenthesis</td>
<td>&lt;</td>
<td>less than</td>
</tr>
<tr>
<td>)</td>
<td>right parenthesis</td>
<td>=</td>
<td>equals</td>
</tr>
<tr>
<td>*</td>
<td>asterisk</td>
<td>&gt;</td>
<td>greater than</td>
</tr>
<tr>
<td>+</td>
<td>plus sign</td>
<td>?</td>
<td>question mark</td>
</tr>
<tr>
<td>,</td>
<td>comma</td>
<td>_</td>
<td>underline or</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>underscore</td>
</tr>
<tr>
<td></td>
<td></td>
<td>vertical bar(^1)</td>
<td>^</td>
</tr>
<tr>
<td>!</td>
<td>exclamation mark</td>
<td>[</td>
<td>left bracket</td>
</tr>
<tr>
<td>[</td>
<td>left brace</td>
<td>]</td>
<td>right bracket</td>
</tr>
<tr>
<td>]</td>
<td>right brace</td>
<td>\</td>
<td>reverse solidus or</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>back slash(^2)</td>
</tr>
</tbody>
</table>

\(^1\) Using the vertical bar (|) character might inhibit code portability between IBM relational products. Use the CONCAT operator in place of the || operator.

\(^2\) Some code pages do not have a code point for the reverse solidus (\) character. When entering Unicode string constants, the UESCAPE clause can be used to specify a Unicode escape character other than reverse solidus.

All multi-byte characters are treated as letters, except for the double-byte blank, which is a special character.
Tokens

Tokens are the basic syntactical units of SQL. A token is a sequence of one or more characters. A token cannot contain blank characters, unless it is a string constant or a delimited identifier, which may contain blanks.

Tokens are classified as ordinary or delimiter:

- An ordinary token is a numeric constant, an ordinary identifier, a host identifier, or a keyword.
  
  **Examples**
  ```
  1 .1 +2 SELECT E 3
  ```

- A delimiter token is a string constant, a delimited identifier, an operator symbol, or any of the special characters shown in the syntax diagrams. A question mark is also a delimiter token when it serves as a parameter marker.
  
  **Examples**
  ```
  , 'string' "fld1" = .
  ```

**Spaces**: A space is a sequence of one or more blank characters. Tokens other than string constants and delimited identifiers must not include a space. Any token may be followed by a space. Every ordinary token must be followed by a space or a delimiter token if allowed by the syntax.

**Comments**: SQL comments are either bracketed (introduced by /* and end with */ or simple (introduced by two consecutive hyphens and end with the end of line). Static SQL statements can include host language comments or SQL comments. Comments can be specified wherever a space can be specified, except within a delimiter token or between the keywords EXEC and SQL.

**Case sensitivity**: Any token may include lowercase letters, but a lowercase letter in an ordinary token is folded to uppercase, except for host variables in the C language, which has case-sensitive identifiers. Delimiter tokens are never folded to uppercase. Thus, the statement:
```sql
select * from EMPLOYEE where lastname = 'Smith';
```

is equivalent, after folding, to:
```sql
SELECT * FROM EMPLOYEE WHERE LASTNAME = 'Smith';
```

Multi-byte alphabetic letters are not folded to uppercase. Single-byte characters (a to z) are folded to uppercase.

For characters in Unicode:

- A character is folded to uppercase, if applicable, if the uppercase character in UTF-8 has the same length as the lowercase character in UTF-8. For example, the Turkish lowercase dotless 'ı' is not folded, because in UTF-8, that character has the value X'C4B1', but the uppercase dotless 'İ' has the value X'49'.
- The folding is done in a locale-insensitive manner. For example, the Turkish lowercase dotted 'ı' is folded to the English uppercase (dotless) 'I'.
- Both halfwidth and fullwidth alphabetic letters are folded to uppercase. For example, the fullwidth lowercase 'a' (U+FF41) is folded to the fullwidth uppercase 'A' (U+FF21).
An identifier is a token that is used to form a name. An identifier in an SQL statement is either an SQL identifier or a host identifier.

- **SQL identifiers**
  
  There are two types of SQL identifiers: ordinary and delimited.
  
  - An ordinary identifier is an uppercase letter followed by zero or more characters, each of which is an uppercase letter, a digit, or the underscore character. Note that ordinary identifiers are converted to uppercase. An ordinary identifier should not be a reserved word.
    
    **Examples**
    
    WKLYSAL   WKLY_SAL
  
  - A delimited identifier is a sequence of one or more characters enclosed by double quotation marks. Two consecutive quotation marks are used to represent one quotation mark within the delimited identifier. In this way an identifier can include lowercase letters.
    
    **Examples**
    
    "WKLY_SAL"   "WKLY SAL"   "UNION"   "wkly_sal"

  Character conversion of identifiers created on a double-byte code page, but used by an application or database on a multi-byte code page, may require special consideration: After conversion, such identifiers may exceed the length limit for an identifier.

- **Host identifiers**

  A host identifier is a name declared in the host program. The rules for forming a host identifier are the rules of the host language. A host identifier should not be greater than 255 bytes in length and should not begin with SQL or DB2 (in uppercase or lowercase characters).

### Naming conventions and implicit object name qualifications

The rules for forming a database object name depend on the type of the object designated by the name. A name may consist of a single SQL identifier or it may be qualified with one or more identifiers that more specifically identify the object. A period must separate each identifier.

The following object names, when used in the context of an SQL procedure, are permitted to use only the characters allowed in an ordinary identifier, even if the names are delimited:

- condition-name
- label
- parameter-name
- procedure-name
- SQL-variable-name
- statement-name

The syntax diagrams use different terms for different types of names. The following list defines these terms.

**alias-name**

A schema-qualified name that designates an alias.
Identifiers

attribute-name
An identifier that designates an attribute of a structured data type.

authorization-name
An identifier that designates a user, group, or role. For a user or a group:
- Valid characters are: ‘A’ through ‘Z’; ‘a’ through ‘z’; ‘0’ through ‘9’; ‘#'; ‘@'; ‘$'; ‘_'; ‘!'; ‘' ('); '{'; '}'; ‘-'; ‘.'; and ‘^’.
- The following characters must be delimited with quotation marks when entered through the command line processor: ‘!'; ‘('); ‘{'; ‘}'; ‘-'; ‘.’; and ‘^’.
- The name must not begin with the characters ‘SYS’, ‘IBM’, or ‘SQL’.
- A delimited authorization ID must not contain lowercase letters.

bufferpool-name
An identifier that designates a buffer pool.

column-name
A qualified or unqualified name that designates a column of a table or view. The qualifier is a table name, a view name, a nickname, or a correlation name.

correlation-name
An identifier that designates a result table.

cursor-name
An identifier that designates an SQL cursor. For host compatibility, a hyphen character may be used in the name.

cursor-type-name
A qualified or unqualified name that designates a user-defined cursor type. An unqualified cursor-type-name in an SQL statement is implicitly qualified, depending on context.

cursor-variable-name
A qualified or unqualified name that designates a global variable, local variable or an SQL parameter of a cursor type. An unqualified cursor variable name in an SQL statement is implicitly qualified, depending on context.

data-source-name
An identifier that designates a data source. This identifier is the first part of a three-part remote object name.

db-partition-group-name
An identifier that designates a database partition group.
Identifiers

descriptor-name
A colon followed by a host identifier that designates an SQL descriptor area (SQLDA). For the description of a host identifier, see "References to host variables" on page 71. Note that a descriptor name never includes an indicator variable.

distinct-type-name
A qualified or unqualified name that designates a distinct type. The unqualified form of distinct-type-name is an SQL identifier. An unqualified distinct type name in an SQL statement is implicitly qualified. The implicit qualifier is a schema name or a module name, which is determined by the context in which distinct-type-name appears. The qualified form is a schema-name followed by a period and an SQL identifier or a module-name (which can also be qualified by a schema-name) followed by a period and an SQL identifier. If the distinct type is defined in a module and used outside of the same module, it must be qualified by the module-name.

event-monitor-name
An identifier that designates an event monitor.

function-mapping-name
An identifier that designates a function mapping.

function-name
A qualified or unqualified name that designates a function. The unqualified form of function-name is an SQL identifier. An unqualified function name in an SQL statement is implicitly qualified. The implicit qualifier is a schema name, which is determined by the context in which the function appears. The qualified form could be is a schema-name followed by a period and an SQL identifier or a module-name followed by a period and an SQL identifier. If the function is published in a module and used outside of the same module, it must be qualified by the module-name.

global-variable-name
A qualified or unqualified name that designates a global variable. An unqualified global variable name in an SQL statement is implicitly qualified, depending on context. If the global variable is defined in a module and used outside of the same module, it must be qualified by the module-name.

group-name
An unqualified identifier that designates a transform group defined for a structured type.

host-variable
A sequence of tokens that designates a host variable. A host variable includes at least one host identifier, explained in "References to host variables" on page 71.

index-name
A schema-qualified name that designates an index or an index specification.

label
An identifier that designates a label in an SQL procedure.

method-name
An identifier that designates a method. The schema context for a method is determined by the schema of the subject type (or a supertype of the subject type) of the method.
Identifiers

module-name
A qualified or unqualified name that designates a module. An unqualified
module-name in an SQL statement is implicitly qualified. The implicit
qualifier is a schema name, which is determined by the context in which
the module-name appears. The qualified form is a schema-name followed
by a period and an SQL identifier.

nickname
A schema-qualified name that designates a federated server reference to a
table or a view.

package-name
A schema-qualified name that designates a package. If a package has a
version ID that is not the empty string, the package name also includes the
version ID at the end of the name, in the form: schema-id.package-
id.version-id.

parameter-name
An identifier that designates a parameter that can be referenced in a
procedure, user-defined function, method, or index extension.

partition-name
An identifier that designates a data partition in a partitioned table.

procedure-name
A qualified or unqualified name that designates a procedure. The
unqualified form of procedure-name is an SQL identifier. An unqualified
procedure name in an SQL statement is implicitly qualified. The implicit
qualifier is a schema name, which is determined by the context in which
the procedure appears. The qualified form is a schema-name followed by a
period and an SQL identifier or a module-name followed by a period and
an SQL identifier. If the procedure is defined in a module and used outside
of the same module, it must be qualified by the module-name.

remote-authorization-name
An identifier that designates a data source user. The rules for authorization
names vary from data source to data source.

remote-function-name
A name that designates a function registered to a data source database.

remote-object-name
A three-part name that designates a data source table or view, and that
identifies the data source in which the table or view resides. The parts of
this name are data-source-name, remote-schema-name, and
remote-table-name.

remote-schema-name
A name that designates the schema to which a data source table or view
belongs. This name is the second part of a three-part remote object name.

remote-table-name
A name that designates a table or view at a data source. This name is the
third part of a three-part remote object name.

remote-type-name
A data type supported by a data source database. Do not use the long
form for built-in types (use CHAR instead of CHARACTER, for example).

role-name
An identifier that designates a role.
Identifiers

row-type-name
A qualified or unqualified name that designates a row type. The unqualified form of row-type-name is an SQL identifier. An unqualified row-type-name in an SQL statement is implicitly qualified. The implicit qualifier is a schema name, which is determined by the context in which the row-type-name appears as described by the rules in "Unqualified user-defined type, function, procedure, specific, global variable and module names" on page 78. The qualified form is a schema-name followed by a period and an SQL identifier.

savepoint-name
An identifier that designates a savepoint.

schema-name
An identifier that provides a logical grouping for SQL objects. A schema name used as a qualifier for the name of an object may be implicitly determined:
• from the value of the CURRENT SCHEMA special register
• from the value of the QUALIFIER precompile/bind option
• on the basis of a resolution algorithm that uses the CURRENT PATH special register
• on the basis of the schema name for another object in the same SQL statement.

To avoid complications, it is recommended that the name SESSION not be used as a schema, except as the schema for declared global temporary tables (which must use the schema name SESSION).

security-label-name
A qualified or unqualified name that designates a security label. An unqualified security label name in an SQL statement is implicitly qualified by the applicable security-policy-name, when one applies. If no security-policy-name is implicitly applicable, the name must be qualified.

security-policy-name
An identifier that designates a security policy.

sequence-name
An identifier that designates a sequence.

server-name
An identifier that designates an application server. In a federated system, the server name also designates the local name of a data source.

specific-name
A qualified or unqualified name that designates a specific name. An unqualified specific name in an SQL statement is implicitly qualified, depending on context.

SQL-variable-name
The name of a local variable in an SQL procedure statement. SQL variable names can be used in other SQL statements where a host variable name is allowed. The name can be qualified by the label of the compound statement that declared the SQL variable.

statement-name
An identifier that designates a prepared SQL statement.
**Identifiers**

**supertype-name**
A qualified or unqualified name that designates the supertype of a type. An unqualified supertype name in an SQL statement is implicitly qualified, depending on context.

**table-name**
A schema-qualified name that designates a table.

**table-reference**
A qualified or unqualified name that designates a table. An unqualified table reference in a common table expression is implicitly qualified by the default schema.

**tablespace-name**
An identifier that designates a table space.

**trigger-name**
A schema-qualified name that designates a trigger.

**type-mapping-name**
An identifier that designates a data type mapping.

**type-name**
A qualified or unqualified name that designates a type. An unqualified type name in an SQL statement is implicitly qualified, depending on context.

**typed-table-name**
A schema-qualified name that designates a typed table.

**typed-view-name**
A schema-qualified name that designates a typed view.

**user-defined-type-name**
A qualified or unqualified name that designates a user-defined data type. The unqualified form of user-defined-type-name is an SQL identifier. An unqualified user-defined-type-name in an SQL statement is implicitly qualified. The implicit qualifier is a schema name or a module name, which is determined by the context in which user-defined-type-name appears. The qualified form is a schema-name followed by a period and an SQL identifier or a module-name (which can also be qualified by a schema-name) followed by a period and an SQL identifier. If the user-defined data type is defined in a module and used outside of the same module, it must be qualified by the module-name.

**view-name**
A schema-qualified name that designates a view.

**wrapper-name**
An identifier that designates a wrapper.

**XML-schema-name**
A qualified or unqualified name that designates an XML schema.

**xsobject-name**
A qualified or unqualified name that designates an object in the XML schema repository.

**Aliases for database objects**
An alias can be thought of as an alternative name for an SQL object. An SQL object, therefore, can be referred to in an SQL statement by its name or by an alias.
A public alias is an alias which can always be referenced without qualifying its name with a schema name. The implicit qualifier of a public alias is SYSPUBLIC, which can also be specified explicitly.

Aliases are also known as synonyms.

An alias can be used wherever the object it is based on can be used. An alias can be created even if the object does not exist (although it must exist by the time a statement referring to it is compiled). It can refer to another alias if no circular or repetitive references are made along the chain of aliases. An alias can only refer to a module, nickname, sequence, table, view, or another alias within the same database. An alias name cannot be used where a new object name is expected, such as in the CREATE TABLE or CREATE VIEW statements; for example, if the table alias name PERSONNEL has been created, subsequent statements such as CREATE TABLE PERSONNEL will return an error.

The option of referring to an object by an alias is not explicitly shown in the syntax diagrams, or mentioned in the descriptions of SQL statements.

A new unqualified alias of a given object type, say for a sequence, cannot have the same fully-qualified name as an existing object of that object type. For example, a sequence alias named ORDERID cannot be defined in the KANDIL schema for the sequence named KANDIL.ORDERID.

The effect of using an alias in an SQL statement is similar to that of text substitution. The alias, which must be defined by the time that the SQL statement is compiled, is replaced at statement compilation time by the qualified object name. For example, if PBIRD.SALES is an alias for DSPN014.DIST4_SALES_148, then at compilation time:

```
SELECT * FROM PBIRD.SALES
```

effectively becomes

```
SELECT * FROM DSPN014.DIST4_SALES_148
```

**Authorization IDs and authorization names**

An authorization ID is a character string that is obtained by the database manager when a connection is established between the database manager and either an application process or a program preparation process. It designates a set of privileges. It may also designate a user or a group of users, but this property is not controlled by the database manager.

Authorization IDs are used by the database manager to provide:

- Authorization checking of SQL statements
- A default value for the QUALIFIER precompile/bind option and the CURRENT SCHEMA special register. The authorization ID is also included in the default CURRENT PATH special register and the FUNCPATH precompile/bind option.

An authorization ID applies to every SQL statement. The authorization ID that applies to a static SQL statement is the authorization ID that is used during program binding. The authorization ID that applies to a dynamic SQL statement is based on the DYNAMICRULES option supplied at bind time, and on the current runtime environment for the package issuing the dynamic SQL statement:

- In a package that has bind behavior, the authorization ID used is the authorization ID of the package owner.
In a package that has define behavior, the authorization ID used is the authorization ID of the corresponding routine’s definer.

In a package that has run behavior, the authorization ID used is the current authorization ID of the user executing the package.

In a package that has invoke behavior, the authorization ID used is the authorization ID currently in effect when the routine is invoked. This is called the runtime authorization ID.

For more information, see “Dynamic SQL characteristics at run time” on page 62.

An authorization name specified in an SQL statement should not be confused with the authorization ID of the statement. An authorization name is an identifier that is used within various SQL statements. An authorization name is used in the CREATE SCHEMA statement to designate the owner of the schema. An authorization name is used in the GRANT and REVOKE statements to designate a target of the grant or revoke operation. Granting privileges to \( X \) means that \( X \) (or a member of the group or role \( X \)) will subsequently be the authorization ID of statements that require those privileges.

Examples:

- Assume that SMITH is the user ID and the authorization ID that the database manager obtained when a connection was established with the application process. The following statement is executed interactively:

  \[
  \text{GRANT SELECT ON TDEPT TO KEENE}
  \]

  SMITH is the authorization ID of the statement. Therefore, in a dynamic SQL statement, the default value of the CURRENT SCHEMA special register is SMITH, and in static SQL, the default value of the QUALIFIER precompile/bind option is SMITH. The authority to execute the statement is checked against SMITH, and SMITH is the \textit{table-name} implicit qualifier based on qualification rules described in “Naming conventions and implicit object name qualifications” on page 54.

  KEENE is an authorization name specified in the statement. KEENE is given the SELECT privilege on SMITH.TDEPT.

- Assume that SMITH has administrative authority and is the authorization ID of the following dynamic SQL statements, with no SET SCHEMA statement issued during the session:

  \[
  \text{DROP TABLE TDEPT}
  \]

  Removes the SMITH.TDEPT table.

  \[
  \text{DROP TABLE SMITH.TDEPT}
  \]

  Removes the SMITH.TDEPT table.

  \[
  \text{DROP TABLE KEENE.TDEPT}
  \]

  Removes the KEENE.TDEPT table. Note that KEENE.TDEPT and SMITH.TDEPT are different tables.

  \[
  \text{CREATE SCHEMA PAYROLL AUTHORIZATION KEENE}
  \]

  KEENE is the authorization name specified in the statement that creates a schema called PAYROLL. KEENE is the owner of the schema PAYROLL and is given CREATEIN, ALTERIN, and DROPIN privileges, with the ability to grant them to others.
Dynamic SQL characteristics at run time

The BIND option DYNAMICRULES determines the authorization ID that is used for checking authorization when dynamic SQL statements are processed. In addition, the option also controls other dynamic SQL attributes, such as the implicit qualifier that is used for unqualified object references, and whether certain SQL statements can be invoked dynamically.

The set of values for the authorization ID and other dynamic SQL attributes is called the dynamic SQL statement behavior. The four possible behaviors are run, bind, define, and invoke. As the following table shows, the combination of the value of the DYNAMICRULES BIND option and the runtime environment determines which of the behaviors is used. DYNAMICRULES RUN, which implies run behavior, is the default.

<table>
<thead>
<tr>
<th>DYNAMICRULES value</th>
<th>Behavior of dynamic SQL statements</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Standalone program environment</td>
</tr>
<tr>
<td>BIND</td>
<td>Bind behavior</td>
</tr>
<tr>
<td>RUN</td>
<td>Run behavior</td>
</tr>
<tr>
<td>DEFINEBIND</td>
<td>Bind behavior</td>
</tr>
<tr>
<td>DEFINERUN</td>
<td>Run behavior</td>
</tr>
<tr>
<td>INVOKEBIND</td>
<td>Bind behavior</td>
</tr>
<tr>
<td>INVOKE_RUN</td>
<td>Run behavior</td>
</tr>
</tbody>
</table>

Run behavior

DB2 uses the authorization ID of the user (the ID that initially connected to DB2) executing the package as the value to be used for authorization checking of dynamic SQL statements and for the initial value used for implicit qualification of unqualified object references within dynamic SQL statements.

Bind behavior

At run time, DB2 uses all the rules that apply to static SQL for authorization and qualification. It takes the authorization ID of the package owner as the value to be used for authorization checking of dynamic SQL statements, and the package default qualifier for implicit qualification of unqualified object references within dynamic SQL statements.

Define behavior

Define behavior applies only if the dynamic SQL statement is in a package that is run within a routine context, and the package was bound with DYNAMICRULES DEFINEBIND or DYNAMICRULES DEFINERUN. DB2 uses the authorization ID of the routine definer (not the routine’s package binder) as the value to be used for authorization checking of dynamic SQL statements, and for implicit qualification of unqualified object references within dynamic SQL statements within that routine.

Invoke behavior

Invoke behavior applies only if the dynamic SQL statement is in a package that is run within a routine context, and the package was bound with DYNAMICRULES INVOKEBIND or DYNAMICRULES INVOKE_RUN. DB2 uses the statement authorization ID in effect when the routine is invoked.
as the value to be used for authorization checking of dynamic SQL, and for implicit qualification of unqualified object references within dynamic SQL statements within that routine. This is summarized by the following table.

<table>
<thead>
<tr>
<th>Invoking Environment</th>
<th>ID Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>any static SQL</td>
<td>implicit or explicit value of the OWNER of the package the SQL invoking the routine came from</td>
</tr>
<tr>
<td>used in definition of view or trigger</td>
<td>definer of the view or trigger</td>
</tr>
<tr>
<td>dynamic SQL from a bind behavior package</td>
<td>implicit or explicit value of the OWNER of the package the SQL invoking the routine came from</td>
</tr>
<tr>
<td>dynamic SQL from a run behavior package</td>
<td>ID used to make the initial connection to DB2</td>
</tr>
<tr>
<td>dynamic SQL from a define behavior package</td>
<td>definer of the routine that uses the package that the SQL invoking the routine came from</td>
</tr>
<tr>
<td>dynamic SQL from an invoke behavior package</td>
<td>the current authorization ID invoking the routine</td>
</tr>
</tbody>
</table>

**Restricted statements when run behavior does not apply**

When bind, define, or invoke behavior is in effect, you cannot use the following dynamic SQL statements: GRANT, REVOKE, ALTER, CREATE, DROP, COMMENT, RENAME, SET INTEGRITY, SET EVENT MONITOR STATE; or queries that reference a nickname.

**Considerations regarding the DYNAMICRULES option**

The CURRENT SCHEMA special register cannot be used to qualify unqualified object references within dynamic SQL statements executed from bind, define or invoke behavior packages. This is true even after you issue the SET CURRENT SCHEMA statement to change the CURRENT SCHEMA special register; the register value is changed but not used.

In the event that multiple packages are referenced during a single connection, all dynamic SQL statements prepared by those packages will exhibit the behavior specified by the DYNAMICRULES option for that specific package and the environment in which they are used.

It is important to keep in mind that when a package exhibits bind behavior, the binder of the package should not have any authorities granted that the user of the package should not receive, because a dynamic statement will be using the authorization ID of the package owner. Similarly, when a package exhibits define behavior, the definer of the routine should not have any authorities granted that the user of the package should not receive.

**Authorization IDs and statement preparation**

If the VALIDATE BIND option is specified at bind time, the privileges required to manipulate tables and views must also exist at bind time. If these privileges or the referenced objects do not exist, and the SQLERROR NOPACKAGE option is in effect, the bind operation will be unsuccessful. If the SQLERROR CONTINUE
option is specified, the bind operation will be successful, and any statements in error will be flagged. Any attempt to execute such a statement will result in an error.

If a package is bound with the VALIDATE RUN option, all normal bind processing is completed, but the privileges required to use the tables and views that are referenced in the application need not exist yet. If a required privilege does not exist at bind time, an incremental bind operation is performed whenever the statement is first executed in an application, and all privileges required for the statement must exist. If a required privilege does not exist, execution of the statement is unsuccessful.

Authorization checking at run time is performed using the authorization ID of the package owner.

**Column names**

The meaning of a column name depends on its context. A column name can be used to:

- Declare the name of a column, as in a CREATE TABLE statement.
- Identify a column, as in a CREATE INDEX statement.
- Specify values of the column, as in the following contexts:
  - In an aggregate function, a column name specifies all values of the column in the group or intermediate result table to which the function is applied. For example, MAX(SALARY) applies the function MAX to all values of the column SALARY in a group.
  - In a GROUP BY or ORDER BY clause, a column name specifies all values in the intermediate result table to which the clause is applied. For example, ORDER BY DEPT orders an intermediate result table by the values of the column DEPT.
  - In an expression, a search condition, or a scalar function, a column name specifies a value for each row or group to which the construct is applied. For example, when the search condition CODE = 20 is applied to some row, the value specified by the column name CODE is the value of the column CODE in that row.
- Temporarily rename a column, as in the correlation-clause of a table-reference in a FROM clause.

**Qualified column names**

A qualifier for a column name may be a table, view, nickname, alias, or correlation name.

Whether a column name may be qualified depends on its context:

- Depending on the form of the COMMENT ON statement, a single column name may need to be qualified. Multiple column names must be unqualified.
- Where the column name specifies values of the column, it may be qualified at the user’s option.
- In the assignment-clause of an UPDATE statement, it may be qualified at the user’s option.
- In all other contexts, a column name must not be qualified.
Where a qualifier is optional, it can serve two purposes. They are described under “Column name qualifiers to avoid ambiguity” on page 66 and “Column name qualifiers in correlated references” on page 68.

**Correlation names**

A correlation name can be defined in the FROM clause of a query and in the first clause of an UPDATE or DELETE statement. For example, the clause FROM X.MYTABLE Z establishes Z as a correlation name for X.MYTABLE.

```
FROM X.MYTABLE Z
```

With Z defined as a correlation name for X.MYTABLE, only Z can be used to qualify a reference to a column of that instance of X.MYTABLE in that SELECT statement.

A correlation name is associated with a table, view, nickname, alias, nested table expression, table function, or data change table reference only within the context in which it is defined. Hence, the same correlation name can be defined for different purposes in different statements, or in different clauses of the same statement.

As a qualifier, a correlation name can be used to avoid ambiguity or to establish a correlated reference. It can also be used merely as a shorter name for a table reference. In the example, Z might have been used merely to avoid having to enter X.MYTABLE more than once.

If a correlation name is specified for a table, view, nickname, or alias name, any qualified reference to a column of that instance of the table, view, nickname, or alias must use the correlation name, rather than the table, view, nickname, or alias name. For example, the reference to EMPLOYEE.PROJECT in the following example is incorrect, because a correlation name has been specified for EMPLOYEE:

**Example**

```
FROM EMPLOYEE E
  WHERE EMPLOYEE.PROJECT='ABC'  * incorrect*
```

The qualified reference to PROJECT should instead use the correlation name, "E", as shown below:

```
FROM EMPLOYEE E
  WHERE E.PROJECT='ABC'
```

Names specified in a FROM clause are either exposed or non-exposed. A table, view, nickname, or alias name is said to be exposed in the FROM clause if a correlation name is not specified. A correlation name is always an exposed name. For example, in the following FROM clause, a correlation name is specified for EMPLOYEE but not for DEPARTMENT, so DEPARTMENT is an exposed name, and EMPLOYEE is not:

```
FROM EMPLOYEE E, DEPARTMENT
```

A table, view, nickname, or alias name that is exposed in a FROM clause may be the same as any other table name, view name or nickname exposed in that FROM clause or any correlation name in the FROM clause. This may result in ambiguous column name references which returns an error (SQLSTATE 42702).
Identifiers

The first two FROM clauses shown below are correct, because each one contains no more than one reference to EMPLOYEE that is exposed:

1. Given the FROM clause:
   
   ```
   FROM EMPLOYEE E1, EMPLOYEE
   ```
   
   a qualified reference such as EMPLOYEE.PROJECT denotes a column of the second instance of EMPLOYEE in the FROM clause. A qualified reference to the first instance of EMPLOYEE must use the correlation name “E1” (E1.PROJECT).

2. Given the FROM clause:
   
   ```
   FROM EMPLOYEE, EMPLOYEE E2
   ```
   
   a qualified reference such as EMPLOYEE.PROJECT denotes a column of the first instance of EMPLOYEE in the FROM clause. A qualified reference to the second instance of EMPLOYEE must use the correlation name “E2” (E2.PROJECT).

3. Given the FROM clause:
   
   ```
   FROM EMPLOYEE, EMPLOYEE
   ```
   
   the two exposed table names included in this clause (EMPLOYEE and EMPLOYEE) are the same. This is allowed, but references to specific column names would be ambiguous (SQLSTATE 42702).

4. Given the following statement:
   
   ```
   SELECT *
   FROM EMPLOYEE E1, EMPLOYEE E2 * incorrect *
   WHERE EMPLOYEE.PROJECT = 'ABC'
   ```
   
   the qualified reference EMPLOYEE.PROJECT is incorrect, because both instances of EMPLOYEE in the FROM clause have correlation names. Instead, references to PROJECT must be qualified with either correlation name (E1.PROJECT or E2.PROJECT).

5. Given the FROM clause:
   
   ```
   FROM EMPLOYEE, X.EMPLOYEE
   ```
   
   a reference to a column in the second instance of EMPLOYEE must use X.EMPLOYEE (X.EMPLOYEE.PROJECT). If X is the CURRENT SCHEMA special register value in dynamic SQL or the QUALIFIER precompile/bind option in static SQL, then the columns cannot be referenced since any such reference would be ambiguous.

The use of a correlation name in the FROM clause also allows the option of specifying a list of column names to be associated with the columns of the result table. As with a correlation name, these listed column names become the exposed names of the columns that must be used for references to the columns throughout the query. If a column name list is specified, then the column names of the underlying table become non-exposed.

Given the FROM clause:

```
FROM DEPARTMENT D (NUM,NAME,MGR,ANUM,LOC)
```

a qualified reference such as D.NUM denotes the first column of the DEPARTMENT table that is defined in the table as DEPTNO. A reference to D.DEPTNO using this FROM clause is incorrect since the column name DEPTNO is a non-exposed column name.

**Column name qualifiers to avoid ambiguity**

In the context of a function, a GROUP BY clause, ORDER BY clause, an expression, or a search condition, a column name refers to values of a column in some table,
view, nickname, nested table expression or table function. The tables, views, nicknames, nested table expressions and table functions that might contain the column are called the object tables of the context. Two or more object tables might contain columns with the same name; one reason for qualifying a column name is to designate the table from which the column comes. Qualifiers for column names are also useful in SQL procedures to distinguish column names from SQL variable names used in SQL statements.

A nested table expression or table function will consider table-references that precede it in the FROM clause as object tables. The table-references that follow are not considered as object tables.

Table designators

A qualifier that designates a specific object table is called a table designator. The clause that identifies the object tables also establishes the table designators for them. For example, the object tables of an expression in a SELECT clause are named in the FROM clause that follows it:

```
SELECT CORZ.COLA, OWNY.MYTABLE.COLA
FROM OXNX.MYTABLE CORZ, OWNY.MYTABLE
```

Table designators in the FROM clause are established as follows:

- A name that follows a table, view, nickname, alias, nested table expression or table function is both a correlation name and a table designator. Thus, CORZ is a table designator. CORZ is used to qualify the first column name in the select list.
- An exposed table, view name, nickname or alias is a table designator. Thus, OWNY.MYTABLE is a table designator. OWNY.MYTABLE is used to qualify the second column name in the select list.

When qualifying a column with the exposed table name form of a table designator, either the qualified or unqualified form of the exposed table name can be used. If the qualified form is used, the qualifier must be the same as the default qualifier for the exposed table name.

For example, assume that the current schema is CORPDATA.

```
SELECT CORPDATA.EMPLOYEE.WORKDEPT FROM EMPLOYEE
```

is valid because the EMPLOYEE table referenced in the FROM clause fully qualifies to CORPDATA.EMPLOYEE, which matches the qualifier for the WORKDEPT column.

```
SELECT EMPLOYEE.WORKDEPT, REGEMP.WORKDEPT
FROM CORPDATA.EMPLOYEE, REGION.EMPLOYEE REGEMP
```

is also valid, because the first select list column references the unqualified exposed table designator CORPDATA.EMPLOYEE, which is in the FROM clause, and the second select list column references the correlation name REGEMP of the table object REGION.EMPLOYEE, which is also in the FROM clause.

Now assume that the current schema is REGION.

```
SELECT CORPDATA.EMPLOYEE.WORKDEPT FROM EMPLOYEE
```

is not valid because the EMPLOYEE table referenced in the FROM clause fully qualifies to REGION.EMPLOYEE, and the qualifier for the WORKDEPT column represents the CORPDATA.EMPLOYEE table.
Identifiers

Each table designator should be unique within a particular FROM clause to avoid the possibility of ambiguous references to columns.

Avoiding undefined or ambiguous references

When a column name refers to values of a column, exactly one object table must include a column with that name. The following situations are considered errors:

- No object table contains a column with the specified name. The reference is undefined.
- The column name is qualified by a table designator, but the table designated does not include a column with the specified name. Again the reference is undefined.
- The name is unqualified, and more than one object table includes a column with that name. The reference is ambiguous.
- The column name is qualified by a table designator, but the table designated is not unique in the FROM clause and both occurrences of the designated table include the column. The reference is ambiguous.
- The column name is in a nested table expression which is not preceded by the TABLE keyword or in a table function or nested table expression that is the right operand of a right outer join or a full outer join and the column name does not refer to a column of a table-reference within the nested table expression's fullselect. The reference is undefined.

Avoid ambiguous references by qualifying a column name with a uniquely defined table designator. If the column is contained in several object tables with different names, the table names can be used as designators. Ambiguous references can also be avoided without the use of the table designator by giving unique names to the columns of one of the object tables using the column name list following the correlation name.

When qualifying a column with the exposed table name form of a table designator, either the qualified or unqualified form of the exposed table name may be used. However, the qualifier used and the table used must be the same after fully qualifying the table name, view name or nickname and the table designator.

1. If the authorization ID of the statement is CORPDATA:
   
   ```sql
   SELECT CORPDATA.EMPLOYEE.WORKDEPT
   FROM EMPLOYEE
   ```

   is a valid statement.

2. If the authorization ID of the statement is REGION:

   ```sql
   SELECT CORPDATA.EMPLOYEE.WORKDEPT
   FROM EMPLOYEE
   ```

   is invalid, because EMPLOYEE represents the table REGION.EMPLOYEE, but the qualifier for WORKDEPT represents a different table, CORPDATA.EMPLOYEE.

Column name qualifiers in correlated references

A fullselect is a form of a query that may be used as a component of various SQL statements. A fullselect used within a search condition of any statement is called a subquery. A fullselect used to retrieve a single value as an expression within a statement is called a scalar fullselect or scalar subquery. A fullselect used in the FROM clause of a query is called a nested table expression. Subqueries in search conditions, scalar subqueries and nested table expressions are referred to as subqueries through the remainder of this topic.
A subquery may include subqueries of its own, and these may, in turn, include subqueries. Thus an SQL statement may contain a hierarchy of subqueries. Those elements of the hierarchy that contain subqueries are said to be at a higher level than the subqueries they contain.

Every element of the hierarchy contains one or more table designators. A subquery can reference not only the columns of the tables identified at its own level in the hierarchy, but also the columns of the tables identified previously in the hierarchy, back to the highest level of the hierarchy. A reference to a column of a table identified at a higher level is called a correlated reference.

For compatibility with existing standards for SQL, both qualified and unqualified column names are allowed as correlated references. However, it is good practice to qualify all column references used in subqueries; otherwise, identical column names may lead to unintended results. For example, if a table in a hierarchy is altered to contain the same column name as the correlated reference and the statement is prepared again, the reference will apply to the altered table.

When a column name in a subquery is qualified, each level of the hierarchy is searched, starting at the same subquery as the qualified column name appears and continuing to the higher levels of the hierarchy until a table designator that matches the qualifier is found. Once found, it is verified that the table contains the given column. If the table is found at a higher level than the level containing column name, then it is a correlated reference to the level where the table designator was found. A nested table expression must be preceded with the optional TABLE keyword in order to search the hierarchy above the fullselect of the nested table expression.

When the column name in a subquery is not qualified, the tables referenced at each level of the hierarchy are searched, starting at the same subquery where the column name appears and continuing to higher levels of the hierarchy, until a match for the column name is found. If the column is found in a table at a higher level than the level containing column name, then it is a correlated reference to the level where the table containing the column was found. If the column name is found in more than one table at a particular level, the reference is ambiguous and considered an error.

In either case, T, used in the following example, refers to the table designator that contains column C. A column name, T.C (where T represents either an implicit or an explicit qualifier), is a correlated reference if, and only if, these conditions are met:

- T.C is used in an expression of a subquery.
- T does not designate a table used in the from clause of the subquery.
- T designates a table used at a higher level of the hierarchy that contains the subquery.

Since the same table, view or nickname can be identified at many levels, unique correlation names are recommended as table designators. If T is used to designate a table at more than one level (T is the table name itself or is a duplicate correlation name), T.C refers to the level where T is used that most directly contains the subquery that includes T.C. If a correlation to a higher level is needed, a unique correlation name must be used.

The correlated reference T.C identifies a value of C in a row or group of T to which two search conditions are being applied: condition 1 in the subquery, and condition

Identifiers
Identifiers

2 at some higher level. If condition 2 is used in a WHERE clause, the subquery is evaluated for each row to which condition 2 is applied. If condition 2 is used in a HAVING clause, the subquery is evaluated for each group to which condition 2 is applied.

For example, in the following statement, the correlated reference X.WORKDEPT (in the last line) refers to the value of WORKDEPT in table EMPLOYEE at the level of the first FROM clause. (That clause establishes X as a correlation name for EMPLOYEE.) The statement lists employees who make less than the average salary for their department.

```
SELECT EMPNO, LASTNAME, WORKDEPT
FROM EMPLOYEE X
WHERE SALARY < (SELECT AVG(SALARY)
                 FROM EMPLOYEE
                 WHERE WORKDEPT = X.WORKDEPT)
```

The next example uses THIS as a correlation name. The statement deletes rows for departments that have no employees.

```
DELETE FROM DEPARTMENT THIS
WHERE NOT EXISTS(SELECT *
                  FROM EMPLOYEE
                  WHERE WORKDEPT = THIS.DEPTNO)
```

References to variables

A variable in an SQL statement specifies a value that can be changed when the SQL statement is executed. There are several types of variables used in SQL statements:

host variable
Host variables are defined by statements of a host language. For more information about how to refer to host variables, see "References to host variables" on page 71.

transition variable
Transition variables are defined in a trigger and refer to either the old or new values of columns. For more information about how to refer to transition variables, see “CREATE TRIGGER statement” in the SQL Reference, Volume 2.

SQL variable
SQL variables are defined by an SQL compound statement in an SQL function, SQL method, SQL procedure, trigger, or dynamic SQL statement. For more information about SQL variables, see “References to SQL parameters, SQL variables, and global variables” in the SQL Reference, Volume 2.

global variable
Global variables are defined by the CREATE VARIABLE statement. For more information about global variables, see “CREATE VARIABLE” and “References to SQL parameters, SQL variables, and global variables” in the SQL Reference, Volume 2.

module variable
Module variables are defined by the ALTER MODULE statement using the ADD VARIABLE or PUBLISH VARIABLE operation. For more information about module variables, see “ALTER MODULE” in the SQL Reference, Volume 2.

SQL parameter
SQL parameters are defined by a CREATE FUNCTION, CREATE
Identifiers

METHOD, or CREATE PROCEDURE statement. For more information about SQL parameters, see “References to SQL parameters, SQL variables, and global variables” in the SQL Reference, Volume 2.

Parameter marker

Parameter markers are specified in a dynamic SQL statement where host variables would be specified if the statement were a static SQL statement. An SQL descriptor or parameter binding is used to associate a value with a parameter marker during dynamic SQL statement processing. For more information about parameter markers, see “Parameter markers” in the SQL Reference, Volume 2.

References to host variables

A host variable is either:

- A variable in a host language such as a C variable, a C++ variable, a COBOL data item, a FORTRAN variable, or a Java variable

or:

- A host language construct that was generated by an SQL precompiler from a variable declared using SQL extensions

that is referenced in an SQL statement. Host variables are either directly defined by statements in the host language or are indirectly defined using SQL extensions.

A host variable in an SQL statement must identify a host variable described in the program according to the rules for declaring host variables.

All host variables used in an SQL statement must be declared in an SQL DECLARE section in all host languages except REXX. No variables may be declared outside an SQL DECLARE section with names identical to variables declared inside an SQL DECLARE section. An SQL DECLARE section begins with BEGIN DECLARE SECTION and ends with END DECLARE SECTION.

The meta-variable host-variable, as used in the syntax diagrams, shows a reference to a host variable. A host-variable as the target variable in a SET variable statement or in the INTO clause of a FETCH, SELECT INTO, or VALUES INTO statement, identifies a host variable to which a value from a column of a row or an expression is assigned. In all other contexts a host-variable specifies a value to be passed to the database manager from the application program.

The meta-variable host-variable in syntax diagrams can generally be expanded to:

```
---:host-identifier
     [INDICATOR]:host-identifier
```

Each host-identifier must be declared in the source program. The variable designated by the second host-identifier must have a data type of small integer.

The first host-identifier designates the main variable. Depending on the operation, it either provides a value to the database manager or is provided a value from the database manager. An input host variable provides a value in the runtime application code page. An output host variable is provided a value that, if necessary, is converted to the runtime application code page when the data is
Identifiers

copied to the output application variable. A given host variable can serve as both an input and an output variable in the same program.

The second host-identifier designates its *indicator variable*. The purposes of the indicator variable are to:

- Specify the null value. A negative value of the indicator variable specifies the null value. A value of -2 indicates a numeric conversion or arithmetic expression error occurred in deriving the result.
- Record the original length of a truncated string (if the source of the value is not a large object type).
- Record the seconds portion of a time if the time is truncated on assignment to a host variable.

For example, if :HV1:HV2 is used to specify an insert or update value, and if HV2 is negative, the value specified is the null value. If HV2 is not negative the value specified is the value of HV1.

Similarly, if :HV1:HV2 is specified in an INTO clause of a FETCH, SELECT INTO, or VALUES INTO statement, and if the value returned is null, HV1 is not changed, and HV2 is set to a negative value. If the database is configured with `dt_sqlmathwarn` yes (or was during binding of a static SQL statement), HV2 could be -2. If HV2 is -2, a value for HV1 could not be returned because of an error converting to the numeric type of HV1, or an error evaluating an arithmetic expression that is used to determine the value for HV1. When accessing a database with a client version earlier than DB2 Universal Database Version 5, HV2 will be -1 for arithmetic exceptions. If the value returned is not null, that value is assigned to HV1 and HV2 is set to zero (unless the assignment to HV1 requires string truncation of a non-LOB string; in which case HV2 is set to the original length of the string). If an assignment requires truncation of the seconds part of a time, HV2 is set to the number of seconds.

If the second host identifier is omitted, the host-variable does not have an indicator variable. The value specified by the host-variable reference :HV1 is always the value of HV1, and null values cannot be assigned to the variable. Thus, this form should not be used in an INTO clause unless the corresponding column cannot contain null values. If this form is used and the column contains nulls, the database manager will generate an error at run time.

An SQL statement that references host variables must be within the scope of the declaration of those host variables. For host variables referenced in the SELECT statement of a cursor, that rule applies to the OPEN statement rather than to the DECLARE CURSOR statement.

**Example**

Using the PROJECT table, set the host variable PNAME (VARCHAR(26)) to the project name (PROJNAME), the host variable STAFF (DECIMAL(5,2)) to the mean staffing level (PRSTAFF), and the host variable MAJPROJ (CHAR(6)) to the major project (MAJPROJ) for project (PROJNO) 'IF1000'. Columns PRSTAFF and MAJPROJ may contain null values, so provide indicator variables STAFF_IND (SMALLINT) and MAJPROJ_IND (SMALLINT).

```
SELECT PROJNAME, PRSTAFF, MAJPROJ
FROM PROJECT
WHERE PROJNO = 'IF1000'
```
**MBCS Considerations**: Whether multi-byte characters can be used in a host variable name depends on the host language.

**Variables in dynamic SQL**

In dynamic SQL statements, parameter markers are used instead of host variables. A parameter marker represents a position in a dynamic SQL statement where the application will provide a value; that is, where a host variable would be found if the statement string were a static SQL statement. The following example shows a static SQL statement using host variables:

```
INSERT INTO DEPARTMENT VALUES (:HV_DEPTNO, :HV_DEPTNAME, :HV_MGRNO, :HV_ADMRDEPT)
```

This example shows a dynamic SQL statement using unnamed parameter markers:

```
INSERT INTO DEPARTMENT VALUES (?, ?, ?, ?)
```

This example shows a dynamic SQL statement using named parameter markers:

```
INSERT INTO DEPARTMENT VALUES (:DEPTNO, :DEPTNAME, :MGRNO, :ADMRDEPT)
```

Named parameter markers can be used to improve the readability of dynamic statement. Although named parameter markers look like host variables, named parameter markers have no associated value and therefore a value must be provided for the parameter marker when the statement is executed. If the INSERT statement using named parameter markers has been prepared and given the prepared statement name of DYNSTMT, then values can be provided for the parameter markers using the following statement:

```
EXECUTE DYNSTMT USING :HV_DEPTNO, :HV_DEPTNAME :HV_MGRNO, :HV_ADMRDEPT
```

This same EXECUTE statement could be used if the INSERT statement using unnamed parameter markers had been prepared and given the prepared statement name of DYNSTMT.

**References to LOB variables**

Regular BLOB, CLOB, and DBCLOB variables, LOB locator variables (see “References to LOB locator variables” on page 74), and LOB file reference variables (see “References to LOB file reference variables” on page 74) can be defined in all host languages. Where LOBs are allowed, the term host-variable in a syntax diagram can refer to a regular host variable, a locator variable, or a file reference variable. Since these are not native data types, SQL extensions are used and the precompilers generate the host language constructs necessary to represent each variable. In the case of REXX, LOBs are mapped to strings.

It is sometimes possible to define a large enough variable to hold an entire large object value. If this is true and if there is no performance benefit to be gained by deferred transfer of data from the server, a locator is not needed. However, since host language or space restrictions will often dictate against storing an entire large object in temporary storage at one time and/or because of performance benefit, a large object may be referenced via a locator and portions of that object may be selected into or updated from host variables that contain only a portion of the large object at one time.
References to LOB locator variables

A locator variable is a host variable that contains the locator representing a LOB value on the application server.

A locator variable in an SQL statement must identify a locator variable described in the program according to the rules for declaring locator variables. This is always indirectly through an SQL statement.

The term locator variable, as used in the syntax diagrams, shows a reference to a locator variable. The meta-variable locator-variable can be expanded to include a host-identifier the same as that for host-variable.

As with all other host variables, a large object locator variable may have an associated indicator variable. Indicator variables for large object locator host variables behave in the same way as indicator variables for other data types. When a null value is returned from the database, the indicator variable is set and the locator host variable is unchanged. This means a locator can never point to a null value.

If a locator-variable that does not currently represent any value is referenced, an error is raised (SQLSTATE 0F001).

At transaction commit, or any transaction termination, all locators acquired by that transaction are released.

References to LOB file reference variables

BLOB, CLOB, and DBCLOB file reference variables are used for direct file input and output for LOBs, and can be defined in all host languages. Since these are not native data types, SQL extensions are used and the precompilers generate the host language constructs necessary to represent each variable. In the case of REXX, LOBs are mapped to strings.

A file reference variable represents (rather than contains) the file, just as a LOB locator represents, rather than contains, the LOB bytes. Database queries, updates and inserts may use file reference variables to store or to retrieve single column values.

A file reference variable has the following properties:

Data Type

BLOB, CLOB, or DBCLOB. This property is specified when the variable is declared.

Direction

This must be specified by the application program at run time (as part of the File Options value). The direction is one of:

- Input (used as a source of data on an EXECUTE statement, an OPEN statement, an UPDATE statement, an INSERT statement, or a DELETE statement).
- Output (used as the target of data on a FETCH statement or a SELECT INTO statement).

File name

This must be specified by the application program at run time. It is one of:

- The complete path name of the file (which is advised).
A relative file name. If a relative file name is provided, it is appended to the current path of the client process.

Within an application, a file should only be referenced in one file reference variable.

File Name Length
This must be specified by the application program at run time. It is the length of the file name (in bytes).

File Options
An application must assign one of a number of options to a file reference variable before it makes use of that variable. Options are set by an INTEGER value in a field in the file reference variable structure. One of the following values must be specified for each file reference variable:

- Input (from client to server)
  **SQL_FILE_READ**
  This is a regular file that can be opened, read and closed. (The option is SQL-FILE-READ in COBOL, sql_file_read in FORTRAN, and READ in REXX.)

- Output (from server to client)
  **SQL_FILE_CREATE**
  Create a new file. If the file already exists, an error is returned. (The option is SQL-FILE-CREATE in COBOL, sql_file_create in FORTRAN, and CREATE in REXX.)

  **SQL_FILE_OVERWRITE (Overwrite)**
  If an existing file with the specified name exists, it is overwritten; otherwise a new file is created. (The option is SQL-FILE-OVERWRITE in COBOL, sql_file_overwrite in FORTRAN, and OVERWRITE in REXX.)

  **SQL_FILE_APPEND**
  If an existing file with the specified name exists, the output is appended to it; otherwise a new file is created. (The option is SQL-FILE-APPEND in COBOL, sql_file_append in FORTRAN, and APPEND in REXX.)

Data Length
This is unused on input. On output, the implementation sets the data length to the length of the new data written to the file. The length is in bytes.

As with all other host variables, a file reference variable may have an associated indicator variable.

### Example of an output file reference variable (in C)

Given a declare section coded as:

```sql
EXEC SQL BEGIN DECLARE SECTION
  SQL TYPE IS CLOB_FILE  hv_text_file;
  char  hv_patent_title[64];
EXEC SQL END DECLARE SECTION
```

Following preprocessing this would be:

```sql
EXEC SQL BEGIN DECLARE SECTION
/* SQL TYPE IS CLOB_FILE  hv_text_file; */
struct {
```
Identifiers

unsigned long name_length; // File Name Length
unsigned long data_length; // Data Length
unsigned long file_options; // File Options
char name[255]; // File Name
} hv_text_file;
char hv_patent_title[64];
EXEC SQL END DECLARE SECTION

Then, the following code can be used to select from a CLOB column in the database into a new file referenced by :hv_text_file.

strcpy(hv_text_file.name, "/u/gainer/papers/sigmod.94");
hv_text_file.name_length = strlen("/u/gainer/papers/sigmod.94");
hv_text_file.file_options = SQL_FILE_CREATE;

EXEC SQL SELECT content INTO :hv_text_file FROM papers WHERE TITLE = 'The Relational Theory behind Juggling';

Example of an input file reference variable (in C)

Given the same declare section as above, the following code can be used to insert the data from a regular file referenced by :hv_text_file into a CLOB column.

strcpy(hv_text_file.name, "/u/gainer/patents/chips.13");
hv_text_file.name_length = strlen("/u/gainer/patents/chips.13");
hv_text_file.file_options = SQL_FILE_READ;
strcpy(hv_text_title, "A Method for Pipelining Chip Consumption");

EXEC SQL INSERT INTO patents( title, text )
VALUES( :hv_patent_title, :hv_text_file );

References to structured type host variables

Structured type variables can be defined in all host languages except FORTRAN, REXX, and Java. Since these are not native data types, SQL extensions are used and the precompilers generate the host language constructs necessary to represent each variable.

As with all other host variables, a structured type variable may have an associated indicator variable. Indicator variables for structured type host variables behave in the same way as indicator variables for other data types. When a null value is returned from the database, the indicator variable is set and the structured type host variable is unchanged.

The actual host variable for a structured type is defined as a built-in data type. The built-in data type associated with the structured type must be assignable:

- from the result of the FROM SQL transform function for the structured type as defined by the specified TRANSFORM GROUP option of the precompile command; and
- to the parameter of the TO SQL transform function for the structured type as defined by the specified TRANSFORM GROUP option of the precompile command.

If using a parameter marker instead of a host variable, the appropriate parameter type characteristics must be specified in the SQLDA. This requires a "doubled" set of SQLVAR structures in the SQLDA, and the SQLDATATYPE_NAME field of the secondary SQLVAR must be filled with the schema and type name of the structured type. If the schema is omitted in the SQLDA structure, an error results (SQLSTATE 07002).
Example

Define the host variables `hv_poly` and `hv_point` (of type POLYGON, using built-in type BLOB(1048576)) in a C program.

```sql
EXEC SQL BEGIN DECLARE SECTION;
  static SQL
  TYPE IS POLYGON AS BLOB(1M)
  hv_poly, hv_point;
EXEC SQL END DECLARE SECTION;
```

SQL path

The SQL path is an ordered list of schema names. The database manager uses the SQL path to resolve the schema name for unqualified data type names (both built-in types and distinct types), global variable names, module names, function names, and procedure names that appear in any context other than as the main object of a CREATE, DROP, COMMENT, GRANT or REVOKE statement. For details, see “Qualification of unqualified object names”.

For example, if the SQL path is SYSIBM. SYSFUN, SYSPROC, SYSIBMADM, SMITH, XGRAPHICS2 and an unqualified distinct type name MYTYPE was specified, the database manager looks for MYTYPE first in schema SYSIBM, then SYSFUN, then SYSPROC, then SYSIBMADM, then SMITH, and then XGRAPHICS2.

The SQL path used depends on the SQL statement:

- For static SQL statements (except for a CALL variable statement), the SQL path used is the SQL path specified when the containing package, procedure, function, trigger, or view was created.
- For dynamic SQL statements (and for a CALL variable statement), the SQL path is the value of the CURRENT PATH special register. CURRENT PATH can be set by the SET PATH statement.

If the SQL path is not explicitly specified, the SQL path is the system path followed by the authorization ID of the statement.

Qualification of unqualified object names

Unqualified object names are implicitly qualified. The rules for qualifying a name differ depending on the type of object that the name identifies.

Unqualified alias, index, package, sequence, table, trigger, and view names

Unqualified alias, index, package, sequence, table, trigger, and view names are implicitly qualified by the default schema.

For static SQL statements, the default schema is the default schema specified when the containing function, package, procedure, or trigger was created.

For dynamic SQL statements, the default schema is the default schema specified for the application process. The default schema can be specified for the application process by using the SET SCHEMA statement. If the default schema is not explicitly specified, the default schema is the authorization ID of the statement.
Unqualified user-defined type, function, procedure, specific, global variable and module names

The qualification of data type (both built-in types and distinct types), global variable, module, function, procedure, and specific names depends on the SQL statement in which the unqualified name appears:

- If an unqualified name is the main object of a CREATE, ALTER, COMMENT, DROP, GRANT, or REVOKE statement, the name is implicitly qualified using the same rules as for qualifying unqualified table names (See “Unqualified alias, index, package, sequence, table, trigger, and view names” on page 77). The main object of an ADD, COMMENT, DROP, or PUBLISH operation of the ALTER MODULE statement must be specified without any qualifier.
- If the context of the reference is within a module, the database manager searches the module for the object, applying the appropriate resolution for the type of object to find a match. If no match is found, the search continues as specified in the next bullet.
- Otherwise, the implicit schema name is determined as follows:
  - For distinct type names, the database manager searches the SQL path and selects the first schema in the SQL path such that the data type exists in the schema.
  - For global variables, the database manager searches the SQL path and selects the first schema in the SQL path such that the global variable exists in the schema.
  - For procedure names, the database manager uses the SQL path in conjunction with procedure resolution.
  - For function names, the database manager uses the SQL path in conjunction with function resolution.
  - For specific names specified for sourced functions, see “CREATE FUNCTION (Sourced)”.

Resolving qualified object names

Objects that are defined in a module that are available for use outside the module must be qualified by the module name. Since a module is a schema object that can also be implicitly qualified, the published module objects can be qualified using an unqualified module name or a schema-qualified module name. When an unqualified module name is used, the reference to the module object appears the same as a schema-qualified object that is not part of a module. Within a specific scope, such as a compound SQL statement, a two-part identifier could also be:

- a column name qualified by a table name
- a row field name qualified by a variable name
- a variable name qualified by a label
- a routine parameter name qualified by a routine name

These objects are resolved within their scope, before considering either schema objects or module object. The following process is used to resolve objects with two-part identifiers that could be a schema object or a module object.

- If the context of the reference is within a module and the qualifier matches the module name, the database manager searches the module for the object, applying the appropriate resolution for the type of object to find a match among published and unpublished module objects. If no match is found, the search continues as specified in the next bullets.
Identifiers

- Assume that the qualifier is a schema name and, if the schema exists, resolve the object in the schema.
- If the qualifier is not an existing schema or the object is not found in the schema that matches the qualifier and the qualifier did not match the context module name, search for the first module that matches the qualifier in the schemas on the SQL path. If authorized to the matching module, resolve to the object in that module, considering only published module objects.
- If the qualifier is not found as a module on the SQL path and the qualifier did not match the context module name, check for a module public synonym that matches the qualifier. If found, resolve the object in the module identified by the module public synonym, considering only published module objects.
The smallest unit of data that can be manipulated in SQL is called a value. Values are interpreted according to the data type of their source. Sources include:

- Constants
- Columns
- Functions
- Expressions
- Special registers.
- Variables (such as host variables, SQL variables, global variables, parameter markers, module variable, and parameters of routines)
- Boolean values

DB2 supports a number of built-in data types. It also provides support for user-defined data types. Figure 10 on page 81 shows the supported built-in data types.
All data types include the null value. The null value is a special value that is distinct from all non-null values and thereby denotes the absence of a (non-null) value. Although all data types include the null value, columns defined as NOT NULL cannot contain null values.

Data type list

Figure 10. The DB2 Built-in Data Types
Numbers

The numeric data types are integer, decimal, floating-point, and decimal floating-point.

The numeric data types are categorized as follows:

- Exact numerics: integer and decimal
- Decimal floating-point
- Approximate numerics: floating-point

Integer includes small integer, large integer, and big integer. Integer numbers are exact representations of integers. Decimal numbers are exact representations of numbers with a fixed precision and scale. Integer and decimal numbers are considered exact numeric types.

Decimal floating-point numbers can have a precision of 16 or 34. Decimal floating-point supports both exact representations of real numbers and approximation of real numbers and so is not considered either an exact numeric type or an approximate numeric type.

Floating-point includes single precision and double precision. Floating-point numbers are approximations of real numbers and are considered approximate numeric types.

All numbers have a sign, a precision, and a scale. For all numbers except decimal floating-point, if a column value is zero, the sign is positive. Decimal floating-point numbers include negative and positive zeros. Decimal floating-point has distinct values for a number and the same number with various exponents (for example: 0.0, 0.00, 0.0E5, 1.0, 1.00, 1.0000). The precision is the total number of decimal digits, excluding the sign. The scale is the total number of decimal digits to the right of the decimal point. If there is no decimal point, the scale is zero.

See also the data type section in the description of the CREATE TABLE statement.

Small integer (SMALLINT)

A small integer is a two-byte integer with a precision of 5 digits. The range of small integers is -32,768 to 32,767.

Large integer (INTEGER)

A large integer is a four-byte integer with a precision of 10 digits. The range of large integers is -2,147,483,648 to +2,147,483,647.

Big integer (BIGINT)

A big integer is an eight-byte integer with a precision of 19 digits. The range of big integers is -9,223,372,036,854,775,808 to +9,223,372,036,854,775,807.

Decimal (DECIMAL or NUMERIC)

A decimal value is a packed decimal number with an implicit decimal point. The position of the decimal point is determined by the precision and the scale of the
number. The scale, which is the number of digits in the fractional part of the number, cannot be negative or greater than the precision. The maximum precision is 31 digits.

All values in a decimal column have the same precision and scale. The range of a decimal variable or the numbers in a decimal column is \(-n\) to \(+n\), where the absolute value of \(n\) is the largest number that can be represented with the applicable precision and scale. The maximum range is \(-10^{31} + 1\) to \(10^{31} - 1\).

**Single-precision floating-point (REAL)**

A single-precision floating-point number is a 32-bit approximation of a real number. The number can be zero or can range from \(-3.4028234663852886e+38\) to \(-1.1754943508222875e-38\), or from \(1.1754943508222875e-38\) to \(3.4028234663852886e+38\).

**Double-precision floating-point (DOUBLE or FLOAT)**

A double-precision floating-point number is a 64-bit approximation of a real number. The number can be zero or can range from \(-1.7976931348623158e+308\) to \(-2.2250738585072014e-308\), or from \(2.2250738585072014e-308\) to \(1.7976931348623158e+308\).

**Decimal floating-point (DECFLOAT)**

A decimal floating-point value is an IEEE 754r number with a decimal point. The position of the decimal point is stored in each decimal floating-point value. The maximum precision is 34 digits. The range of a decimal floating-point number is either 16 or 34 digits of precision, and an exponent range of \(10^{-383}\) to \(10^{384}\) or \(10^{-6143}\) to \(10^{6144}\), respectively. The minimum exponent, \(E_{\text{min}}\) for DECFLOAT values is \(-383\) for DECFLOAT(16) and \(-6143\) for DECFLOAT(34). The maximum exponent, \(E_{\text{max}}\) for DECFLOAT values is 384 for DECFLOAT(16) and 6144 for DECFLOAT(34).

In addition to finite numbers, decimal floating-point numbers are able to represent one of the following named decimal floating-point special values:

- **Infinity** - a value that represents a number whose magnitude is infinitely large
- **Quiet NaN** - a value that represents undefined results and that does not cause an invalid number warning
- **Signalling NaN** - a value that represents undefined results and that causes an invalid number warning if used in any numerical operation

When a number has one of these special values, its coefficient and exponent are undefined. The sign of an infinity value is significant, because it is possible to have positive or negative infinity. The sign of a NaN value has no meaning for arithmetic operations.

**Subnormal numbers and underflow**

Nonzero numbers whose adjusted exponents are less than \(E_{\text{min}}\) are called subnormal numbers. These subnormal numbers are accepted as operands for all operations and can result from any operation.

For a subnormal result, the minimum values of the exponent become \(E_{\text{min}} - (\text{precision}-1)\), called \(E_{\text{tiny}}\), where precision is the working precision. If necessary, the result is rounded to ensure that the exponent is no smaller than \(E_{\text{tiny}}\). If the result
becomes inexact during rounding, an underflow warning is returned. A subnormal result does not always return the underflow warning.

When a number underflows to zero during a calculation, its exponent will be $E_{\text{tiny}}$. The maximum value of the exponent is unaffected.

The maximum value of the exponent for subnormal numbers is the same as the minimum value of the exponent that can arise during operations that do not result in subnormal numbers. This occurs when the length of the coefficient in decimal digits is equal to the precision.
Character strings

A character string is a sequence of bytes. The length of the string is the number of bytes in the sequence. If the length is zero, the value is called the empty string. This value should not be confused with the null value.

Fixed-length character string (CHAR)

All values in a fixed-length string column have the same length, which is determined by the length attribute of the column. The length attribute must be between 1 and 254, inclusive.

Varying-length character strings

There are two types of varying-length character strings:
• A VARCHAR value can be up to 32 672 bytes long.
• A CLOB (character large object) value can be up to 2 gigabytes minus 1 byte (2 147 483 647 bytes) long. A CLOB is used to store large SBCS or mixed (SBCS and MBCS) character-based data (such as documents written with a single character set) and, therefore, has an SBCS or mixed code page associated with it.

Special restrictions apply to expressions resulting in a CLOB data type, and to structured type columns; such expressions and columns are not permitted in:
• A SELECT list preceded by the DISTINCT clause
• A GROUP BY clause
• An ORDER BY clause
• A subselect of a set operator other than UNION ALL
• A basic, quantified, BETWEEN, or IN predicate
• An aggregate function
• VARGRAPHIC, TRANSLATE, and datetime scalar functions
• The pattern operand in a LIKE predicate, or the search string operand in a POSSTR function
• The string representation of a datetime value.

The functions in the SYSFUN schema taking a VARCHAR as an argument will not accept VARCHARs greater than 4 000 bytes long as an argument. However, many of these functions also have an alternative signature accepting a CLOB(1M). For these functions, the user may explicitly cast the greater than 4 000 VARCHAR strings into CLOBs and then recast the result back into VARCHARs of desired length.

NUL-terminated character strings found in C are handled differently, depending on the standards level of the precompile option.

Each character string is further defined as one of:

Bit data
Data that is not associated with a code page.

Single-byte character set (SBCS) data
Data in which every character is represented by a single byte.

Mixed data
Data that may contain a mixture of characters from a single-byte character set and a multi-byte character set (MBCS).
Character strings

Note: The LONG VARCHAR data type continues to be supported but is deprecated, not recommended, and might be removed in a future release.

String units in built-in functions

The ability to specify string units for certain built-in functions allows you to process string data in a more "character-based manner" than a "byte-based manner". The string unit determines the length in which an operation is to occur. You can specify CODEUNITS16, CODEUNITS32, or OCTETS as the string unit for an operation.

**CODEUNITS16**
Specifies that Unicode UTF-16 is the unit for the operation. CODEUNITS16 is useful when an application is processing data in code units that are two bytes in width. Note that some characters, known as supplementary characters, require two UTF-16 code units to be encoded. For example, the musical symbol G clef requires two UTF-16 code units (X’D834’ and X’DD1E’ in UTF-16BE).

**CODEUNITS32**
Specifies that Unicode UTF-32 is the unit for the operation. CODEUNITS32 is useful for applications that process data in a simple, fixed-length format, and that must return the same answer regardless of the storage format of the data (ASCII, UTF-8, or UTF-16).

**OCTETS**
Specifies that bytes are the units for the operation. OCTETS is often used when an application is interested in allocating buffer space or when operations need to use simple byte processing.

The calculated length of a string computed using OCTETS (bytes) might differ from that computed using CODEUNITS16 or CODEUNITS32. When using OCTETS, the length of the string is determined by simply counting the number of bytes in the string. When using CODEUNITS16 or CODEUNITS32, the length of the string is determined by counting the number of 16-bit or 32-bit code units necessary to represent the string in UTF-16 or UTF-32, respectively. The length determined using CODEUNITS16 and CODEUNITS32 will be identical unless the data contains supplementary characters (see Difference between CODEUNITS16 and CODEUNITS32 on page 87).

For example, assume that NAME, a VARCHAR(128) column encoded in Unicode UTF-8, contains the value 'Jürgen'. The following two queries, which count the length of the string in CODEUNITS16 and CODEUNITS32, respectively, return the same value (6).

```sql
SELECT CHARACTER_LENGTH(NAME, CODEUNITS16) FROM T1
WHERE NAME = 'Jürgen'
```

```sql
SELECT CHARACTER_LENGTH(NAME, CODEUNITS32) FROM T1
WHERE NAME = 'Jürgen'
```

The next query, which counts the length of the string in OCTETS, returns the value 7.

```sql
SELECT CHARACTER_LENGTH(NAME, OCTETS) FROM T1
WHERE NAME = 'Jürgen'
```

These values represent the length of the string expressed in the specified string unit.
Character strings

The following table shows the UTF-8, UTF-16BE (big-endian), and UTF-32BE (big-endian) representations of the name ‘Jürgen’:

<table>
<thead>
<tr>
<th>Format</th>
<th>Representation of the name ‘Jürgen’</th>
</tr>
</thead>
<tbody>
<tr>
<td>UTF-8</td>
<td>X'4AC3BC7267656E'</td>
</tr>
<tr>
<td>UTF-16BE</td>
<td>X'004A00FC00726765006E'</td>
</tr>
<tr>
<td>UTF-32BE</td>
<td>X'0000004A000000FC0000007267650006E'</td>
</tr>
</tbody>
</table>

The representation of the character ‘ü’ differs among the three string units:
- The UTF-8 representation of the character ‘ü’ is X’C3BC’.
- The UTF-16BE representation of the character ‘ü’ is X’00FC’.
- The UTF-32BE representation of the character ‘ü’ is X’000000FC’.

Specifying string units for a built-in function does not affect the data type or the code page of the result of the function. If necessary, DB2 converts the data to Unicode for evaluation when CODEUNITS16 or CODEUNITS32 is specified.

When OCTETS is specified for the LOCATE or POSITION function, and the code pages of the string arguments differ, DB2 converts the data to the code page of the source-string argument. In this case, the result of the function is in the code page of the source-string argument. When OCTETS is specified for functions that take a single string argument, the data is evaluated in the code page of the string argument, and the result of the function is in the code page of the string argument.

Difference between CODEUNITS16 and CODEUNITS32

When CODEUNITS16 or CODEUNITS32 is specified, the result is the same except when the data contains Unicode supplementary characters. This is because a supplementary character is represented by two UTF-16 code units or one UTF-32 code unit. In UTF-8, a non-supplementary character is represented by 1 to 3 bytes, and a supplementary character is represented by 4 bytes. In UTF-16, a non-supplementary character is represented by one CODEUNITS16 code unit or 2 bytes, and a supplementary character is represented by two CODEUNITS16 code units or 4 bytes. In UTF-32, a character is represented by one CODEUNITS32 code unit or 4 bytes.

For example, the following table shows the hexadecimal values for the mathematical bold capital A and the Latin capital letter A. The mathematical bold capital A is a supplementary character that is represented by 4 bytes in UTF-8, UTF-16, and UTF-32.

<table>
<thead>
<tr>
<th>Character</th>
<th>UTF-8 representation</th>
<th>UTF-16BE representation</th>
<th>UTF-32BE representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unicode value X'1D400' - 'A'; mathematical bold capital A</td>
<td>X'F09D9080'</td>
<td>X'D835DC00'</td>
<td>X'0001D400'</td>
</tr>
<tr>
<td>Unicode value X'0041' - 'A'; latin capital letter A</td>
<td>X'41'</td>
<td>X'0041'</td>
<td>X'00000041'</td>
</tr>
</tbody>
</table>

Assume that C1 is a VARCHAR(128) column, encoded in Unicode UTF-8, and that table T1 contains one row with the value of the mathematical bold capital A (X'F09D9080'). The following queries return different results:
<table>
<thead>
<tr>
<th>Query</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>SELECT CHARACTER_LENGTH(C1, CODEUNITS16) FROM T1</code></td>
<td>2</td>
</tr>
<tr>
<td><code>SELECT CHARACTER_LENGTH(C1, CODEUNITS32) FROM T1</code></td>
<td>1</td>
</tr>
<tr>
<td><code>SELECT CHARACTER_LENGTH(C1, OCTETS) FROM T1</code></td>
<td>4</td>
</tr>
</tbody>
</table>
Graphic strings

A graphic string is a sequence of bytes that represents double-byte character data. The length of the string is the number of double-byte characters in the sequence. If the length is zero, the value is called the empty string. This value should not be confused with the null value.

Graphic strings are not checked to ensure that their values contain only double-byte character code points. (The exception to this rule is an application precompiled with the WCHARTYPE CONVERT option. In this case, validation does occur.) Rather, the database manager assumes that double-byte character data is contained in graphic data fields. The database manager does check that a graphic string value is an even number of bytes long.

NUL-terminated graphic strings found in C are handled differently, depending on the standards level of the precompile option. This data type cannot be created in a table. It can only be used to insert data into and retrieve data from the database.

Fixed-length graphic strings (GRAPHIC)

All values in a fixed-length graphic string column have the same length, which is determined by the length attribute of the column. The length attribute must be between 1 and 127, inclusive.

Varying-length graphic strings

There are two types of varying-length graphic string:

• A VARGRAPHIC value can be up to 16 336 double-byte characters long.
• A DBCLOB (double-byte character large object) value can be up to 1 073 741 823 double-byte characters long. A DBCLOB is used to store large DBCS character-based data (such as documents written with a single character set) and, therefore, has a DBCS code page associated with it.

Special restrictions apply to an expression that results in a varying-length graphic string whose maximum length is greater than 127 bytes. These restrictions are the same as those specified in “Varying-length character strings” on page 85.

Note: The LONG VARGRAPHIC data type continues to be supported but is deprecated, not recommended, and might be removed in a future release.
Binary strings

Binary strings

A binary string is a sequence of bytes. Unlike character strings, which usually contain text data, binary strings are used to hold non-traditional data such as pictures, voice, or mixed media. Character strings of the FOR BIT DATA subtype may be used for similar purposes. Binary strings are not associated with a code page. Binary strings have the same restrictions as character strings (for details, see “Varying-length character strings” on page 85). Only character strings of the FOR BIT DATA subtype are compatible with binary strings.

Binary large object (BLOB)

A binary large object is a varying-length binary string that can be up to 2 gigabytes minus 1 byte (2,147,483,647 bytes) long. BLOBs can hold structured data for exploitation by user-defined types and user-defined functions. Like FOR BIT DATA character strings, BLOB strings are not associated with a code page.
Large objects (LOBs)

The term large object and the generic acronym LOB refer to the BLOB, CLOB, or DBCLOB data type. LOB values are subject to restrictions, as described in “Varying-length character strings” on page 85. These restrictions apply even if the length attribute of the LOB string is 254 bytes or less.

LOB values can be very large, and the transfer of these values from the database server to client application program host variables can be time consuming. Because application programs typically process LOB values one piece at a time, rather than as a whole, applications can reference a LOB value by using a large object locator.

A large object locator, or LOB locator, is a host variable whose value represents a single LOB value on the database server.

An application program can select a LOB value into a LOB locator. Then, using the LOB locator, the application program can request database operations on the LOB value (such as applying the scalar functions SUBSTR, CONCAT, VALUE, or LENGTH; performing an assignment; searching the LOB with LIKE or POSSTR; or applying user-defined functions against the LOB) by supplying the locator value as input. The resulting output (data assigned to a client host variable) would typically be a small subset of the input LOB value.

LOB locators can represent more than just base values; they can also represent the value associated with a LOB expression. For example, a LOB locator might represent the value associated with:

```sql
SUBSTR(<lob 1> CONCAT <lob 2> CONCAT <lob 3>, <start>, <length>)
```

When a null value is selected into a normal host variable, the indicator variable is set to -1, signifying that the value is null. In the case of LOB locators, however, the meaning of indicator variables is slightly different. Because a locator host variable can itself never be null, a negative indicator variable value indicates that the LOB value represented by the LOB locator is null. The null information is kept local to the client by virtue of the indicator variable value — the server does not track null values with valid locators.

It is important to understand that a LOB locator represents a value, not a row or a location in the database. Once a value is selected into a locator, there is no operation that one can perform on the original row or table that will affect the value which is referenced by the locator. The value associated with a locator is valid until the transaction ends, or until the locator is explicitly freed, whichever comes first. Locators do not force extra copies of the data to provide this function. Instead, the locator mechanism stores a description of the base LOB value. The materialization of the LOB value (or expression, as shown above) is deferred until it is actually assigned to some location — either a user buffer in the form of a host variable, or another record in the database.

A LOB locator is only a mechanism used to refer to a LOB value during a transaction; it does not persist beyond the transaction in which it was created. It is not a database type; it is never stored in the database and, as a result, cannot participate in views or check constraints. However, because a LOB locator is a client representation of a LOB type, there are SQLTYPEs for LOB locators so that they can be described within an SQLDA structure used by FETCH, OPEN, or EXECUTE statements.
Datetime values

The datetime data types include DATE, TIME, and TIMESTAMP. Although datetime values can be used in certain arithmetic and string operations, and are compatible with certain strings, they are neither strings nor numbers.

Date

A date is a three-part value (year, month, and day). The range of the year part is 0001 to 9999. The range of the month part is 1 to 12. The range of the day part is 1 to \( x \), where \( x \) depends on the month.

The internal representation of a date is a string of 4 bytes. Each byte consists of 2 packed decimal digits. The first 2 bytes represent the year, the third byte the month, and the last byte the day.

The length of a DATE column, as described in the SQLDA, is 10 bytes, which is the appropriate length for a character string representation of the value.

Time

A time is a three-part value (hour, minute, and second) designating a time of day under a 24-hour clock. The range of the hour part is 0 to 24. The range of the other parts is 0 to 59. If the hour is 24, the minute and second specifications are zero.

The internal representation of a time is a string of 3 bytes. Each byte consists of 2 packed decimal digits. The first byte represents the hour, the second byte the minute, and the last byte the second.

The length of a TIME column, as described in the SQLDA, is 8 bytes, which is the appropriate length for a character string representation of the value.

Timestamp

A timestamp is a six or seven-part value (year, month, day, hour, minute, second, and optional fractional seconds) designating a date and time as defined above, except that the time could also include an additional part designating a fraction of a second. The number of digits in the fractional seconds is specified using an attribute in the range from 0 to 12 with a default of 6.

The internal representation of a timestamp is a string of between 7 and 13 bytes. Each byte consists of 2 packed decimal digits. The first 4 bytes represent the date, the next 3 bytes the time, and the last 0 to 6 bytes the fractional seconds.

The length of a TIMESTAMP column, as described in the SQLDA, is between 19 and 32 bytes, which is the appropriate length for the character string representation of the value.

String representations of datetime values

Values whose data types are DATE, TIME, or TIMESTAMP are represented in an internal form that is transparent to the user. Date, time, and timestamp values can, however, also be represented by strings. This is useful because there are no constants or variables whose data types are DATE, TIME, or TIMESTAMP. Before it can be retrieved, a datetime value must be assigned to a string variable. The CHAR function or the GRAPHIC function (for Unicode databases only) can be
used to change a datetime value to a string representation. The string representation is normally the default format of datetime values associated with the territory code of the application, unless overridden by specification of the DATETIME option when the program is precompiled or bound to the database.

No matter what its length, a large object string cannot be used as a string representation of a datetime value (SQLSTATE 42884).

When a valid string representation of a datetime value is used in an operation with an internal datetime value, the string representation is converted to the internal form of the date, time, or timestamp value before the operation is performed.

Date, time and timestamp strings must contain only characters and digits.

**Date strings**

A string representation of a date is a string that starts with a digit and has a length of at least 8 characters. Trailing blanks may be included; leading zeros may be omitted from the month and day portions.

Valid string formats for dates are listed in the following table. Each format is identified by name and associated abbreviation.

<table>
<thead>
<tr>
<th>Format Name</th>
<th>Abbreviation</th>
<th>Date Format</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>International Standards Organization</td>
<td>ISO</td>
<td>yyyy-mm-dd</td>
<td>1991-10-27</td>
</tr>
<tr>
<td>IBM USA standard</td>
<td>USA</td>
<td>mm/dd/yyyy</td>
<td>10/27/1991</td>
</tr>
<tr>
<td>IBM European standard</td>
<td>EUR</td>
<td>dd.mm.yyyy</td>
<td>27.10.1991</td>
</tr>
<tr>
<td>Japanese Industrial Standard Christian Era</td>
<td>JIS</td>
<td>yyyy-mm-dd</td>
<td>1991-10-27</td>
</tr>
<tr>
<td>Site-defined</td>
<td>LOC</td>
<td>Depends on the territory code of the application</td>
<td>—</td>
</tr>
</tbody>
</table>

**Time strings**

A string representation of a time is a string that starts with a digit and has a length of at least 4 characters. Trailing blanks can be included; a leading zero can be omitted from the hour part of the time, and seconds can be omitted entirely. If seconds are omitted, an implicit specification of 0 seconds is assumed. Thus, 13:30 is equivalent to 13:30:00.

Valid string formats for times are listed in the following table. Each format is identified by name and associated abbreviation.

<table>
<thead>
<tr>
<th>Format Name</th>
<th>Abbreviation</th>
<th>Time Format</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>International Standards Organization</td>
<td>ISO</td>
<td>hh.mm.ss</td>
<td>13.30.05</td>
</tr>
<tr>
<td>IBM USA standard</td>
<td>USA</td>
<td>hh:mm AM or PM</td>
<td>1:30 PM</td>
</tr>
<tr>
<td>IBM European standard</td>
<td>EUR</td>
<td>hh.mm.ss</td>
<td>13.30.05</td>
</tr>
</tbody>
</table>
Datetime values

Table 9. Formats for String Representations of Times (continued)

<table>
<thead>
<tr>
<th>Format Name</th>
<th>Abbreviation</th>
<th>Time Format</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Japanese Industrial Standard</td>
<td>JIS</td>
<td>hh:mm:ss</td>
<td>13:30:05</td>
</tr>
<tr>
<td>Christian Era</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Site-defined</td>
<td>LOC</td>
<td>Depends on the territory code of the application</td>
<td>—</td>
</tr>
</tbody>
</table>

Note:

1. In ISO, EUR, or JIS format, .ss (or :ss) is optional.
2. The International Standards Organization changed the time format so that it is identical to the Japanese Industrial Standard Christian Era format. Therefore, use the JIS format if an application requires the current International Standards Organization format.
3. In the USA time string format, the minutes specification can be omitted, indicating an implicit specification of 00 minutes. Thus, 1 PM is equivalent to 1:00 PM.
4. In the USA time string format, the hour must not be greater than 12 and cannot be 0, except in the special case of 00:00 AM. There is a single space before ‘AM’ or ‘PM’. ‘AM’ and ‘PM’ can be represented in lowercase or uppercase characters.

Using the JIS format of the 24-hour clock, the correspondence between the USA format and the 24-hour clock is as follows:

- 12:01 AM through 12:59 AM corresponds to 00:01:00 through 00:59:00.
- 01:00 AM through 11:59 AM corresponds to 01:00:00 through 11:59:00.
- 12:00 PM (noon) through 11:59 PM corresponds to 12:00:00 through 23:59:00.
- 12:00 AM (midnight) corresponds to 24:00:00 and 00:00 AM (midnight) corresponds to 00:00:00.

Timestamp strings

A string representation of a timestamp is a string that starts with a digit and has a length of at least 16 characters. The complete string representation of a timestamp has the form yyyy-mm-dd-hh.mm.ss or yyyy-mm-dd-hh.mm.ss.nnnnnnnnnnn, where the number of digits for fractional seconds can range from 0 to 12. Trailing blanks may be included. Leading zeros may be omitted from the month, day, and hour part of the timestamp. Trailing zeros can be truncated or entirely omitted from the fractional seconds. If a string representation of a timestamp is implicitly cast to a value with a TIMESTAMP data type, the timestamp precision of the result of the cast is determined by the precision of the TIMESTAMP operand in an expression or the precision of the TIMESTAMP target in an assignment. Digits in the string beyond the timestamp precision of the cast are truncated or any missing digits to reach the timestamp precision of the cast are assumed to be zeroes. For example, 1991-3-2-8.30.00 is equivalent to 1991-03-02-08.30.00.000000000000.

A string representation of a timestamp can be given a different timestamp precision by explicitly casting the value to a timestamp with a specified precision. If the string is a constant, an alternative is to precede the string constant with the TIMESTAMP keyword. For example, TIMESTAMP '2007-03-28 14:50:35.123' has the TIMESTAMP(3) data type.
SQL statements also support the ODBC string representation of a timestamp, but as an input value only. The ODBC string representation of a timestamp has the form `yyyy-mm-dd hh:mm:ss.nnnnnnnnnnnn`, where the number of digits for fractional seconds can range from 0 to 12.

**Datetime values**

SQL statements also support the ODBC string representation of a timestamp, but as an input value only. The ODBC string representation of a timestamp has the form `yyyy-mm-dd hh:mm:ss.nnnnnnnnnnnn`, where the number of digits for fractional seconds can range from 0 to 12.
Boolean values

A Boolean value represents a truth value of TRUE or FALSE. A Boolean expression or predicate can result in a value of unknown, which is represented as the null value.

The BOOLEAN type is a built-in data type that can only be used as the data type of:

- A local variable in a compound SQL (compiled) statement
- A parameter of an SQL routine
- The returns type of an SQL function
- A global variable

A variable or parameter defined with the BOOLEAN type can only be used in compound SQL (compiled) statements.
Cursor values

A cursor value is used to represent a reference to an underlying cursor.

The CURSOR type is a built-in data type that can only be used as the data type of:
- A local variable in a compound SQL (compiled) statement
- A parameter of an SQL routine
- The returns type of an SQL function
- A global variable

A variable or parameter defined with the CURSOR type can only be used in compound SQL (compiled) statements.

A cursor variable is an SQL variable, SQL parameter, or global variable of a cursor type. A cursor variable is said to have an underlying cursor that corresponds to the cursor created for a SELECT statement and assigned to that variable. More than one cursor variable may share the same underlying cursor.

Cursor variables can be used the same way as conventional SQL cursors to iterate through a result set of a SELECT statement with OPEN, FETCH, and CLOSE statements.
XML values

XML values

An XML value represents well-formed XML in the form of an XML document, XML content, or a sequence of XML nodes. An XML value that is stored in a table as a value of a column defined with the XML data type must be a well-formed XML document. XML values are processed in an internal representation that is not comparable to any string value. An XML value can be transformed into a serialized string value representing the XML document using the XMLSERIALIZE function. Similarly, a string value that represents an XML document can be transformed into an XML value using the XMLPARSE function. An XML value can be implicitly parsed or serialized when exchanged with application string and binary data types.

Special restrictions apply to expressions that result in an XML data type value; such expressions and columns are not permitted (SQLSTATE 42818):

- A SELECT list preceded by the DISTINCT clause
- A GROUP BY clause
- An ORDER BY clause
- A subselect of a set operator other than UNION ALL
- A basic, quantified, BETWEEN, IN, or LIKE predicate
- An aggregate function with DISTINCT
Array values

An array is a structure that contains an ordered collection of data elements in which each element can be referenced by its index value in the collection. The cardinality of an array is the number of elements in the array. All elements in an array have the same data type.

An ordinary array has a defined upper bound on the number of elements, known as the maximum cardinality. Each element in the array is referenced by its ordinal position as the index value. If \( N \) is the number of elements in an ordinary array, the ordinal position associated with each element is an integer value greater than or equal to 1 and less than or equal to \( N \).

An associative array has no specific upper bound on the number of elements. Each element is referenced by its associated index value. The data type of the index value can be an integer or a character string but is the same data type for the entire array.

The maximum cardinality of an ordinary array is not related to its physical representation, unlike the maximum cardinality of arrays in programming languages such as C. Instead, the maximum cardinality is used by the system at run time to ensure that subscripts are within bounds. The amount of memory required to represent an ordinary array value is not proportional to the maximum cardinality of its type.

The amount of memory required to represent an array value is usually proportional to its cardinality. When an array is being referenced, all of the values in the array are stored in main memory. Therefore, arrays that contain a large amount of data will consume large amounts of main memory.
Anchored types

Anchored types
An anchored type defines a data type based on another SQL object such as a column, global variable, SQL variable, SQL parameter, or the row of a table or view.

A data type defined using an anchored type definition maintains a dependency on the object to which it is anchored. Any change in the data type of the anchor object will impact the anchored data type. If anchored to the row of a table or view, the anchored data type is ROW with the fields defined by the columns of the anchor table or anchor view.
User-defined types

There are six types of user-defined data type:

- Distinct type
- Structured type
- Reference type
- Array type
- Row type
- Cursor type

Each of these types is described in the following sections.

Distinct type

A distinct type is a user-defined data type that shares its internal representation with an existing type (its “source” type), but is considered to be a separate and incompatible type for most operations. For example, one might want to define a picture type, a text type, and an audio type, all of which have quite different semantics, but which use the built-in data type BLOB for their internal representation.

The following example illustrates the creation of a distinct type named AUDIO:

```
CREATE TYPE AUDIO AS BLOB (1M)
```

Although AUDIO has the same representation as the built-in data type BLOB, it is considered to be a separate type; this allows the creation of functions written specifically for AUDIO, and assures that these functions will not be applied to values of any other data type (pictures, text, and so on).

Distinct types have qualified identifiers. If the schema name is not used to qualify the distinct type name when used in other than the CREATE TYPE (Distinct), DROP, or COMMENT statements, the SQL path is searched in sequence for the first schema with a distinct type that matches.

Distinct types support strong typing by ensuring that only those functions and operators explicitly defined on a distinct type can be applied to its instances. For this reason, a distinct type does not automatically acquire the functions and operators of its source type, because these may not be meaningful. (For example, the LENGTH function of the AUDIO type might return the length of its object in seconds rather than in bytes.)

Distinct types sourced on LOB types are subject to the same restrictions as their source type.

However, certain functions and operators of the source type can be explicitly specified to apply to the distinct type. This can be done by creating user-defined functions that are sourced on functions defined on the source type of the distinct type. The comparison operators are automatically generated for user-defined distinct types, except those using BLOB, CLOB, or DBCLOB as the source type. In addition, functions are generated to support casting from the source type to the distinct type, and from the distinct type to the source type.
User-defined types

Structured type

A structured type is a user-defined data type that has a structure that is defined in the database. It contains a sequence of named attributes, each of which has a data type. A structured type also includes a set of method specifications.

A structured type may be used as the type of a table, view, or column. When used as a type for a table or view, that table or view is known as a typed table or typed view, respectively. For typed tables and typed views, the names and data types of the attributes of the structured type become the names and data types of the columns of this typed table or typed view. Rows of the typed table or typed view can be thought of as a representation of instances of the structured type. When used as a data type for a column, the column contains values of that structured type (or values of any of that type’s subtypes, as defined below). Methods are used to retrieve or manipulate attributes of a structured column object.

Terminology: A supertype is a structured type for which other structured types, called subtypes, have been defined. A subtype inherits all the attributes and methods of its supertype and may have additional attributes and methods defined. The set of structured types that are related to a common supertype is called a type hierarchy and the type that does not have any supertype is called the root type of the type hierarchy.

The term subtype applies to a user-defined structured type and all user-defined structured types that are below it in the type hierarchy. Therefore, a subtype of a structured type T is T and all structured types below T in the hierarchy. A proper subtype of a structured type T is a structured type below T in the type hierarchy.

There are restrictions on having recursive type definitions in a type hierarchy. For this reason, it is necessary to develop a shorthand way of referring to the specific type of recursive definitions that are allowed. The following definitions are used:

- **Directly uses**: A type A is said to directly use another type B, if and only if one of the following is true:
  1. type A has an attribute of type B
  2. type B is a subtype of A, or a supertype of A

- **Indirectly uses**: A type A is said to indirectly use a type B, if one of the following is true:
  1. type A directly uses type B
  2. type A directly uses some type C, and type C indirectly uses type B

A type may not be defined so that one of its attribute types directly or indirectly uses itself. If it is necessary to have such a configuration, consider using a reference as the attribute. For example, with structured type attributes, there cannot be an instance of "employee" with an attribute of "manager" when "manager" is of type "employee". There can, however, be an attribute of "manager" with a type of REF(employee).

A type cannot be dropped if certain other objects use the type, either directly or indirectly. For example, a type cannot be dropped if a table or view column makes direct or indirect use of the type.
User-defined types

Reference type

A reference type is a companion type to a structured type. Similar to a distinct type, a reference type is a scalar type that shares a common representation with one of the built-in data types. This same representation is shared for all types in the type hierarchy. The reference type representation is defined when the root type of a type hierarchy is created. When using a reference type, a structured type is specified as a parameter of the type. This parameter is called the target type of the reference.

The target of a reference is always a row in a typed table or a typed view. When a reference type is used, it may have a scope defined. The scope identifies a table (called the target table) or view (called the target view) that contains the target row of a reference value. The target table or view must have the same type as the target type of the reference type. An instance of a scoped reference type uniquely identifies a row in a typed table or typed view, called the target row.

Array type

A user-defined array type is a data type that is defined as an array with elements of another data type. Every ordinary array type has an index with the data type of INTEGER and has a defined maximum cardinality. Every associative array has an index with the data type of INTEGER or VARCHAR and does not have a defined maximum cardinality.

Row type

A row type is a data type that is defined as an ordered sequence of named fields, each with an associated data type, which effectively represents a row. A row type can be used as the data type for variables and parameters in SQL PL to provide simple manipulation of a row of data.

Cursor data type

A user-defined cursor type is a user-defined data type defined with the keyword CURSOR and optionally with an associated row type. A user-defined cursor type with an associated row type is a strongly-typed cursor type; otherwise, it is a weakly-typed cursor type. A value of a user-defined cursor type represents a reference to an underlying cursor.
Promotion of data types

Promotion of data types

Data types can be classified into groups of related data types. Within such groups, a precedence order exists where one data type is considered to precede another data type. This precedence is used to allow the promotion of one data type to a data type later in the precedence ordering. For example, the data type CHAR can be promoted to VARCHAR; INTEGER can be promoted to DOUBLE-PRECISION; but CLOB is NOT promotable to VARCHAR.

Promotion of data types is used when:
• Performing function resolution
• Casting user-defined types
• Assigning user-defined types to built-in data types

Table 10 shows the precedence list (in order) for each data type and can be used to determine the data types to which a given data type can be promoted. The table shows that the best choice is always the same data type instead of choosing to promote to another data type.

Table 10. Data Type Precedence Table

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Data Type Precedence List (in best-to-worst order)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMALLINT</td>
<td>SMALLINT, INTEGER, BIGINT, decimal, real, double, DECFLOAT</td>
</tr>
<tr>
<td>INTEGER</td>
<td>INTEGER, BIGINT, decimal, real, double, DECFLOAT</td>
</tr>
<tr>
<td>BIGINT</td>
<td>BIGINT, decimal, real, double, DECFLOAT</td>
</tr>
<tr>
<td>decimal</td>
<td>decimal, real, double, DECFLOAT</td>
</tr>
<tr>
<td>real</td>
<td>real, double, DECFLOAT</td>
</tr>
<tr>
<td>double</td>
<td>double, DECFLOAT</td>
</tr>
<tr>
<td>DECFLOAT</td>
<td>DECFLOAT</td>
</tr>
<tr>
<td>CHAR</td>
<td>CHAR, VARCHAR, CLOB</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>VARCHAR, CLOB</td>
</tr>
<tr>
<td>CLOB</td>
<td>CLOB</td>
</tr>
<tr>
<td>GRAPHIC</td>
<td>GRAPHIC, VARGRAPHIC, DBCLOB</td>
</tr>
<tr>
<td>VARGRAPHIC</td>
<td>VARGRAPHIC, DBCLOB</td>
</tr>
<tr>
<td>DBCLOB</td>
<td>DBCLOB</td>
</tr>
<tr>
<td>BLOB</td>
<td>BLOB</td>
</tr>
<tr>
<td>DATE</td>
<td>DATE, TIMESTAMP</td>
</tr>
<tr>
<td>TIME</td>
<td>TIME</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>BOOLEAN</td>
<td>BOOLEAN</td>
</tr>
<tr>
<td>CURSOR</td>
<td>CURSOR</td>
</tr>
<tr>
<td>ARRAY</td>
<td>ARRAY</td>
</tr>
<tr>
<td>udt</td>
<td>udt (same name) or a supertype of udt</td>
</tr>
<tr>
<td>REF(T)</td>
<td>REF(S) (provided that S is a supertype of T)</td>
</tr>
<tr>
<td>ROW</td>
<td>ROW</td>
</tr>
</tbody>
</table>
Table 10. Data Type Precedence Table (continued)

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Data Type Precedence List (in best-to-worst order)</th>
</tr>
</thead>
</table>

Note:

1. The lowercase types above are defined as follows:
   - decimal = DECIMAL(p,s) or NUMERIC(p,s)
   - real = REAL or FLOAT(n), where n is not greater than 24
   - double = DOUBLE, DOUBLE-PRECISION, FLOAT or FLOAT(n), where n is greater than 24
   - udt = a user-defined type

   Shorter and longer form synonyms of the listed data types are considered to be the same as the listed form.

2. For a Unicode database, the following are considered to be equivalent data types:
   - CHAR and GRAPHIC
   - VARCHAR and VARGRAPHIC
   - CLOB and DBCLOB

   When resolving a function within a Unicode database, if a user-defined function and a built-in function are both applicable for a given function invocation, then generally the built-in function will be invoked. The UDF will be invoked only if its schema precedes SYSIBM in the CURRENT PATH special register and if its argument data types match all the function invocation argument data types, regardless of Unicode data type equivalence.
Casting between data types

There are many occasions where a value with a given data type needs to be cast to a different data type or to the same data type with a different length, precision, or scale. Data type promotion is one example where the promotion of one data type to another data type requires that the value be cast to the new data type. A data type that can be cast to another data type is castable from the source data type to the target data type.

The casting of one data type to another can occur implicitly or explicitly. The cast functions, CAST specification, or XMLCAST specification can be used to explicitly change a data type, depending on the data types involved. In addition, when a sourced user-defined function is created, the data types of the parameters of the source function must be castable to the data types of the function that is being created.

The supported casts between built-in data types are shown in Table 11 on page 108. The first column represents the data type of the cast operand (source data type), and the data types across the top represent the target data type of the cast operation. A 'Y' indicates that the CAST specification can be used for the combination of source and target data types. Cases in which only the XMLCAST specification can be used are noted.

If truncation occurs when any data type is cast to a character or graphic data type, a warning is returned if any non-blank characters are truncated. This truncation behavior is unlike the assignment to a character or graphic data type, when an error occurs if any non-blank characters are truncated.

The following casts involving distinct types are supported (using the CAST specification unless noted otherwise):

- Cast from distinct type DT to its source data type S
- Cast from the source data type S of distinct type DT to distinct type DT
- Cast from distinct type DT to the same distinct type DT
- Cast from a data type A to distinct type DT where A is promotable to the source data type S of distinct type DT
- Cast from an INTEGER to distinct type DT with a source data type SMALLINT
- Cast from a DOUBLE to distinct type DT with a source data type REAL
- Cast from a DECFLOAT to distinct type DT with a source data type of DECFLOAT
- Cast from a VARCHAR to distinct type DT with a source data type CHAR
- Cast from a VARGRAPHIC to distinct type DT with a source data type GRAPHIC
- For a Unicode database, cast from a VARCHAR or a VARGRAPHIC to distinct type DT with a source data type CHAR or GRAPHIC
- Cast from a distinct type DT with a source data type S to XML using the XMLCAST specification
- Cast from an XML to a distinct type DT with a source data type of any built-in data type, using the XMLCAST specification depending on the XML schema data type of the XML value

FOR BIT DATA character types cannot be cast to CLOB.
Casting between data types

For casts that involve an array type as a target, the data type of the elements of the source array value must be castable to the data type of the elements of the target array data (SQLSTATE 42846). If the target array type is an ordinary array, the source array value must be an ordinary array (SQLSTATE 42821) and the cardinality of the source array value must be less than or equal to the maximum cardinality of the target array data type (SQLSTATE 2202F). If the target array type is an associative array, the data type of the index for the source array value must be castable to data type of the index for the target array type. A user-defined array type value can only be cast to another user-defined array type with the same name (SQLSTATE 42846).

A cursor type cannot be either the source data type or the target data type of a CAST specification, except to cast a parameter marker to a cursor type.

For casts that involve a row type as a target, the degree of the source row value expression and degree of the target row type must match and each field in the source row value expression must be castable to the corresponding target field. A user-defined row type value can only be cast to another user-defined row-type with the same name (SQLSTATE 42846).

It is not possible to cast a structured type value to something else. A structured type \( ST \) should not need to be cast to one of its supertypes, because all methods on the supertypes of \( ST \) are applicable to \( ST \). If the desired operation is only applicable to a subtype of \( ST \), use the subtype-treatment expression to treat \( ST \) as one of its subtypes.

When a user-defined data type involved in a cast is not qualified by a schema name, the SQL path is used to find the first schema that includes the user-defined data type by that name.

The following casts involving reference types are supported:

- cast from reference type \( RT \) to its representation data type \( S \)
- cast from the representation data type \( S \) of reference type \( RT \) to reference type \( RT \)
- cast from reference type \( RT \) with target type \( T \) to a reference type \( RS \) with target type \( S \) where \( S \) is a supertype of \( T \).
- cast from a data type \( A \) to reference type \( RT \), where \( A \) is promotable to the representation data type \( S \) of reference type \( RT \).

When the target type of a reference data type involved in a cast is not qualified by a schema name, the SQL path is used to find the first schema that includes the user-defined data type by that name.
Casting between data types
Table 11. Supported Casts between Built-in Data Types
Target Data Type

Source Data Type

S
M
A
L
L
I
N
T

I
N
T
E
G
E
R

B
I
G
I
N
T

D
E
C
I
M
A
L

SMALLINT

Y

Y

Y

Y

INTEGER
BIGINT
DECIMAL
REAL

Y
Y
Y
Y

DOUBLE
DECFLOAT
CHAR

Y
Y
Y

Y
Y
Y
Y
Y
Y
Y

Y
Y
Y
Y
Y
Y
Y

Y
Y
Y
Y
Y
Y
Y

C
H
A
R

R
E
A
L

D
O
U
B
L
E

D
E
C
F
L
O
A
T

C
H
A
R

Y

Y

Y

Y

Y
Y
Y
Y
Y
Y
Y

Y
Y
Y
Y
Y
Y
Y

Y
Y
Y
Y
Y
Y
Y

Y
Y
Y
Y
Y
Y
Y

V
A
R
C
H
A
R

V
A
R
G
R
A
P
H
I
C

D
B
C
L
O
B

B
L
O
B

D
A
T
E

-

-

-

F
B
D2

V
A
R
C
H
A
R

F
B
D2

C
L
O
B

G
R
A
P
H
I
C

Y

Y

Y

-

Y1

Y1

-

Y

1

Y

1

Y

1

Y

1

Y

1

Y

1

Y

1

Y

1

Y

1

Y

1

-

Y

1

Y

1

-

Y

1

Y

1

Y
Y
Y
Y
Y
Y
Y

Y
Y
Y
Y
Y
Y
Y

Y
Y
Y
Y
Y
Y
Y

Y

-

-

-

-

-

1

-

-

Y3

-

-

Y

3

-

Y

3

-

Y

3

-

Y

3

-

3

-

-

-

-

X
M
L

B
O
O
L
E
A
N

-

-

-

-

Y

-

T
I
M
E

T
I
M
E
S
T
A
M
P

-

-

-

-

-

-

-

-

-

-

Y

-

-

-

-

-

Y

Y

Y

Y

4

-

3

-

Y

CHAR FOR BIT
DATA

Y

Y

Y

Y

Y

Y

Y

Y

Y

Y

Y

-

-

-

-

Y

Y

Y

Y

Y

VARCHAR

Y

Y

Y

Y

Y

Y

Y

Y

Y

Y

Y

Y

Y1

Y1

Y1

Y

Y

Y

Y

Y4

-

3

-

VARCHAR FOR BIT
DATA

Y

Y

Y

Y

Y

Y

Y

Y

Y

Y

Y

-

-

-

-

Y

Y

Y

Y

Y

CLOB

-

-

-

-

-

-

-

Y

-

Y

-

Y

Y1

Y1

Y1

Y

-

-

-

Y4

-

-

1

-

1

Y

3

-

Y

3

-

3

-

4

-

3

-

3

-

3

GRAPHIC
VARGRAPHIC
DBCLOB
BLOB

Y

1

-

XML

Y

BOOLEAN

-

5

Y
-

SQL Reference, Volume 1

Y

1

5

Y

1

Y

1

-

-

Y

-

Y

1

-

Y

-

TIMESTAMP

Y

1

-

-

TIME

Y

1

-

-

DATE

108

Y

1

Y

Y
Y
Y

5

5

-

Y
-

Y
-

-

Y

1

5

Y

1

-

Y

1

-

Y

Y

Y

Y

Y

1

Y

5

Y

1

-

-

5

Y

1

-

-

-

Y

1

-

-

-

-

Y

1

-

-

Y

Y

1

-

-

Y
Y

Y

1

-

-

Y

Y

1

Y
Y

-

Y

1

-

Y

1

-

Y

Y

Y
Y

Y
Y

Y

Y

Y

Y

5

5

5

5

Y
-

Y
-

Y
-

Y
-

-

-

-

Y
Y
-

-

Y

Y

5

Y
Y
-

Y

1

Y

1

Y

1

Y

5

-

Y

Y
Y
-

Y

1

Y

1

Y

1

Y

5

-

Y

Y

-

-

Y
-

1

Y
-

5

Y

1

Y

1

-

-

Y

1

Y

1

-

-

-

5

Y

Y

-

-

-

Y

Y

-

Y

Y

1

Y

-

-

Y

Y

Y

-

Y
Y

Y

Y

Y

Y

-

5

5

5

Y

-

-

Y7

Y
-

Y
-

Y
-


### Casting between data types

#### Table 11. Supported Casts between Built-in Data Types (continued)

<table>
<thead>
<tr>
<th>Source Data Type</th>
<th>Target Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>V</td>
</tr>
<tr>
<td>M</td>
<td>D</td>
</tr>
<tr>
<td>A</td>
<td>C</td>
</tr>
<tr>
<td>I</td>
<td>R</td>
</tr>
<tr>
<td>N</td>
<td>C</td>
</tr>
<tr>
<td>T</td>
<td>I</td>
</tr>
<tr>
<td>B</td>
<td>G</td>
</tr>
<tr>
<td>E</td>
<td>G</td>
</tr>
<tr>
<td>D</td>
<td>M</td>
</tr>
<tr>
<td>C</td>
<td>B</td>
</tr>
<tr>
<td>A</td>
<td>E</td>
</tr>
<tr>
<td>R</td>
<td>O</td>
</tr>
<tr>
<td>D</td>
<td>T</td>
</tr>
<tr>
<td>E</td>
<td>L</td>
</tr>
</tbody>
</table>

**Notes**

- See the description preceding the table for information on supported casts involving user-defined types and reference types.
- It is not possible to cast a structured type value to anything else.
- The LONG VARCHAR and LONG VARGRAPHIC data types continue to be supported but are deprecated, not recommended, and might be removed in a future release.
- Cast is only supported for Unicode databases.
- FOR BIT DATA
- Cast can only be performed using XMLCAST.
- An XMLPARSE function is implicitly processed to convert a string to XML on assignment (INSERT or UPDATE) of a string to an XML column. The string must be a well-formed XML document for the assignment to succeed.
- Cast can only be performed using XMLCAST and depends on the underlying XML schema data type of the XML value. For details, see "XMLCAST".
- A cursor type cannot be either the source data type or the target data type of a CAST specification, except to cast a parameter marker to a cursor type.
- Only supported using the CAST specification. No cast function exists.

Table [Table 12](#) shows where to find information about the rules that apply when casting to the identified target data types.

#### Table 12. Rules for Casting to a Data Type

<table>
<thead>
<tr>
<th>Target Data Type</th>
<th>Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMALLINT</td>
<td>“SMALLINT scalar function” in SQL Reference, Volume 1</td>
</tr>
<tr>
<td>INTEGER</td>
<td>“INTEGER scalar function” in SQL Reference, Volume 1</td>
</tr>
<tr>
<td>BIGINT</td>
<td>“BIGINT scalar function” in SQL Reference, Volume 1</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>“DECIMAL scalar function” in SQL Reference, Volume 1</td>
</tr>
<tr>
<td>NUMERIC</td>
<td>“DECIMAL scalar function” in SQL Reference, Volume 1</td>
</tr>
<tr>
<td>REAL</td>
<td>“REAL scalar function” in SQL Reference, Volume 1</td>
</tr>
</tbody>
</table>
Casting between data types

Table 12. Rules for Casting to a Data Type (continued)

<table>
<thead>
<tr>
<th>Target Data Type</th>
<th>Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOUBLE</td>
<td>“DOUBLE scalar function” in SQL Reference, Volume 1</td>
</tr>
<tr>
<td>DECFLOAT</td>
<td>“DECFLOAT scalar function” in SQL Reference, Volume 1</td>
</tr>
<tr>
<td>CHAR</td>
<td>“CHAR scalar function” in SQL Reference, Volume 1</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>“VARCHAR scalar function” in SQL Reference, Volume 1</td>
</tr>
<tr>
<td>CLOB</td>
<td>“CLOB scalar function” in SQL Reference, Volume 1</td>
</tr>
<tr>
<td>GRAPHIC</td>
<td>“GRAPHIC scalar function” in SQL Reference, Volume 1</td>
</tr>
<tr>
<td>VARGRAPHIC</td>
<td>“VARGRAPHIC scalar function” in SQL Reference, Volume 1</td>
</tr>
<tr>
<td>DBCLOB</td>
<td>“DBCLOB scalar function” in SQL Reference, Volume 1</td>
</tr>
<tr>
<td>BLOB</td>
<td>“BLOB scalar function” in SQL Reference, Volume 1</td>
</tr>
<tr>
<td>DATE</td>
<td>“DATE scalar function” in SQL Reference, Volume 1</td>
</tr>
<tr>
<td>TIME</td>
<td>“TIME scalar function” in SQL Reference, Volume 1</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>If the source type is a character string, see “TIMESTAMP scalar function” in SQL Reference, Volume 1, where one operand is specified. If the source data type is a DATE, the timestamp is composed of the specified date and a time of 00:00:00.</td>
</tr>
</tbody>
</table>

Casting non-XML values to XML values

Table 13. Supported Casts from Non-XML Values to XML Values

<table>
<thead>
<tr>
<th>Source Data Type</th>
<th>Target Data Type</th>
<th>XML</th>
<th>Resulting XML Schema Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMALLINT</td>
<td>XML</td>
<td>Y</td>
<td>xs:short</td>
</tr>
<tr>
<td>INTEGER</td>
<td>XML</td>
<td>Y</td>
<td>xs:int</td>
</tr>
<tr>
<td>BIGINT</td>
<td>XML</td>
<td>Y</td>
<td>xs:long</td>
</tr>
<tr>
<td>DECIMAL or NUMERIC</td>
<td>XML</td>
<td>Y</td>
<td>xs:decimal</td>
</tr>
<tr>
<td>REAL</td>
<td>XML</td>
<td>Y</td>
<td>xs:float</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>XML</td>
<td>Y</td>
<td>xs:double</td>
</tr>
<tr>
<td>DECFLOAT</td>
<td>XML</td>
<td>N</td>
<td>-</td>
</tr>
<tr>
<td>CHAR</td>
<td>XML</td>
<td>Y</td>
<td>xs:string</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>XML</td>
<td>Y</td>
<td>xs:string</td>
</tr>
<tr>
<td>CLOB</td>
<td>XML</td>
<td>Y</td>
<td>xs:string</td>
</tr>
<tr>
<td>GRAPHIC</td>
<td>XML</td>
<td>Y</td>
<td>xs:string</td>
</tr>
</tbody>
</table>
Table 13. Supported Casts from Non-XML Values to XML Values (continued)

<table>
<thead>
<tr>
<th>Source Data Type</th>
<th>Target Data Type</th>
<th>XML</th>
<th>Resulting XML Schema Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>VARGRAPHIC</td>
<td>XML</td>
<td>Y</td>
<td>xs:string</td>
</tr>
<tr>
<td>DBCLOB</td>
<td>XML</td>
<td>Y</td>
<td>xs:string</td>
</tr>
<tr>
<td>DATE</td>
<td>XML</td>
<td>Y</td>
<td>xs:date</td>
</tr>
<tr>
<td>TIME</td>
<td>XML</td>
<td>Y</td>
<td>xs:time</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>XML</td>
<td>Y</td>
<td>xs:dateTime¹</td>
</tr>
<tr>
<td>BLOB</td>
<td>XML</td>
<td>Y</td>
<td>xs:base64Binary</td>
</tr>
<tr>
<td>character type FOR BIT DATA</td>
<td>XML</td>
<td>Y</td>
<td>xs:base64Binary</td>
</tr>
</tbody>
</table>

Notes

¹ The source data type TIMESTAMP supports timestamp precision of 0 to 12. The maximum fractional seconds precision of xs:dateTime is 6. If the timestamp precision of a TIMESTAMP source data type exceeds 6, the value is truncated when cast to xs:dateTime.

The LONG VARCHAR and LONG VARGRAPHIC data types continue to be supported but are deprecated, not recommended, and might be removed in a future release.

When character string values are cast to XML values, the resulting xs:string atomic value cannot contain illegal XML characters (SQLSTATE 0N002). If the input character string is not in Unicode, the input characters are converted to Unicode.

Casting to SQL binary types results in XQuery atomic values with the type xs:base64Binary.

Casting XML values to non-XML values

An XMLCAST from an XML value to a non-XML value can be described as two casts: an XQuery cast that converts the source XML value to an XQuery type corresponding to the SQL target type, followed by a cast from the corresponding XQuery type to the actual SQL type.

An XMLCAST is supported if the target type has a corresponding XQuery target type that is supported, and if there is a supported XQuery cast from the source value’s type to the corresponding XQuery target type. The target type that is used in the XQuery cast is based on the corresponding XQuery target type and might contain some additional restrictions.

The following table lists the XQuery types that result from such conversion.

Table 14. Supported Casts from XML Values to Non-XML Values

<table>
<thead>
<tr>
<th>Target Data Type</th>
<th>Source Data Type</th>
<th>XML</th>
<th>Corresponding XQuery Target Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMALLINT</td>
<td>XML</td>
<td>Y</td>
<td>xs:short</td>
</tr>
<tr>
<td>INTEGER</td>
<td>XML</td>
<td>Y</td>
<td>xs:int</td>
</tr>
<tr>
<td>BIGINT</td>
<td>XML</td>
<td>Y</td>
<td>xs:long</td>
</tr>
</tbody>
</table>
## Casting between data types

Table 14. Supported Casts from XML Values to Non-XML Values (continued)

<table>
<thead>
<tr>
<th>Target Data Type</th>
<th>Source Data Type</th>
<th>XML</th>
<th>Corresponding XQuery Target Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>DECIMAL or NUMERIC</td>
<td>Y</td>
<td></td>
<td>xs:decimal</td>
</tr>
<tr>
<td>REAL</td>
<td>Y</td>
<td></td>
<td>xs:float</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>Y</td>
<td></td>
<td>xs:double</td>
</tr>
<tr>
<td>DECFLOAT</td>
<td>Y</td>
<td></td>
<td>no matching type(^1)</td>
</tr>
<tr>
<td>CHAR</td>
<td>Y</td>
<td></td>
<td>xs:string</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>Y</td>
<td></td>
<td>xs:string</td>
</tr>
<tr>
<td>CLOB</td>
<td>Y</td>
<td></td>
<td>xs:string</td>
</tr>
<tr>
<td>GRAPHIC</td>
<td>Y</td>
<td></td>
<td>xs:string</td>
</tr>
<tr>
<td>VARGRAPHIC</td>
<td>Y</td>
<td></td>
<td>xs:string</td>
</tr>
<tr>
<td>DBCLOB</td>
<td>Y</td>
<td></td>
<td>xs:string</td>
</tr>
<tr>
<td>DATE</td>
<td>Y</td>
<td></td>
<td>xs:date</td>
</tr>
<tr>
<td>TIME (without time zone)</td>
<td>Y</td>
<td></td>
<td>xs:time</td>
</tr>
<tr>
<td>TIMESTAMP (without time zone)</td>
<td>Y</td>
<td></td>
<td>xs:dateTime(^2)</td>
</tr>
<tr>
<td>BLOB</td>
<td>Y</td>
<td></td>
<td>xs:base64Binary</td>
</tr>
<tr>
<td>CHAR FOR BIT DATA</td>
<td>N</td>
<td></td>
<td>not castable</td>
</tr>
<tr>
<td>VARCHAR FOR BIT DATA</td>
<td>Y</td>
<td></td>
<td>xs:base64Binary</td>
</tr>
</tbody>
</table>

Notes

\(^1\) DB2 supports XML Schema 1.0, which does not provide a matching XML schema type for a DECFLOAT. Processing of the XQuery cast step of XMLCAST is handled as follows:
- If the source value is typed with an XML schema numeric type, use that numeric type.
- If the source value is typed with the XML schema type xs:boolean, use xs:double.
- Otherwise, use xs:string with additional checking for a valid numeric format.

\(^2\) The maximum fractional seconds precision of xs:dateTime is 6. The source data type TIMESTAMP supports timestamp precision of 0 to 12. If the timestamp precision of a TIMESTAMP target data type is less than 6, the value is truncated when cast from xs:dateTime. If the timestamp precision of a TIMESTAMP target data type exceeds 6, the value is padded with zeros when cast from xs:dateTime.

In the following restriction cases, a derived by restriction XML schema data type is effectively used as the target data type for the XQuery cast.
- XML values that are to be converted to string types must fit within the length limits of those DB2 types without truncation of any characters or bytes. The name used for the derived XML schema type is the uppercase SQL type name followed by an underscore character and the maximum length of the string; for example, VARCHAR_20 if the XMLCAST target data type is VARCHAR(20).
- XML values that are to be converted to DECIMAL values must fit within the precision of the specified DECIMAL values, and must not contain more nonzero digits after the decimal point than the scale. The name used for the derived XML schema type is DECIMAL_\(\text{precision}\_\text{scale}\), where \(\text{precision}\) is the precision of...
Casting between data types

the target SQL data type, and scale is the scale of the target SQL data type; for example, DECIMAL_9_2 if the XMLCAST target data type is DECIMAL(9,2).

- XML values that are to be converted to TIME values cannot contain a seconds component with nonzero digits after the decimal point. The name used for the derived XML schema type is TIME.

The derived XML schema type name only appears in a message if an XML value does not conform to one of these restrictions. This type name helps one to understand the error message, and does not correspond to any defined XQuery type. If the input value does not conform to the base type of the derived XML schema type (the corresponding XQuery target type), the error message might indicate that type instead. Because this derived XML schema type name format might change in the future, it should not be used as a programming interface.

Before an XML value is processed by the XQuery cast, any document node in the sequence is removed and each direct child of the removed document node becomes an item in the sequence. If the document node has multiple direct children nodes, the revised sequence will have more items than the original sequence. The XML value without any document nodes is then atomized using the XQuery fn:data function, with the resulting atomized sequence value used in the XQuery cast. If the atomized sequence value is an empty sequence, a null value is returned from the cast without any further processing. If there are multiple items in the atomized sequence value, an error is returned (SQLSTATE 10507).

If the target type of XMLCAST is the SQL data type DATE, TIME, or TIMESTAMP, the resulting XML value from the XQuery cast is also adjusted to UTC, and the time zone component of the value is removed.

When the corresponding XQuery target type value is converted to the SQL target type, binary XML data types, such as xs:base64Binary or xs:hexBinary, are converted from character form to actual binary data.

If an xs:double or xs:float value of INF, -INF, or NaN is cast (using XMLCAST) to an SQL data type DOUBLE or REAL value, an error is returned (SQLSTATE 22003). An xs:double or xs:float value of -0 is converted to +0.

The target type can be a user-defined distinct type if the source operand is not a user-defined distinct type. In this case, the source value is cast to the source type of the user-defined distinct type (that is, the target type) using the XMLCAST specification, and then this value is cast to the user-defined distinct type using the CAST specification.

In a non-Unicode database, casting from an XML value to a non-XML target type involves code page conversion from an internal UTF-8 format to the database code page. This conversion will result in the introduction of substitution characters if any code point in the XML value is not present in the database code page.
Assignments and comparisons

The basic operations of SQL are assignment and comparison. Assignment operations are performed during the execution of INSERT, UPDATE, FETCH, SELECT INTO, VALUES INTO and SET transition-variable statements. Arguments of functions are also assigned when invoking a function. Comparison operations are performed during the execution of statements that include predicates and other language elements such as MAX, MIN, DISTINCT, GROUP BY, and ORDER BY.

One basic rule for both operations is that the data type of the operands involved must be compatible. The compatibility rule also applies to set operations.

Another basic rule for assignment operations is that a null value cannot be assigned to a column that cannot contain null values, nor to a host variable that does not have an associated indicator variable.

Following is a compatibility matrix showing the system-defined data type compatibilities for assignment and comparison operations.

<table>
<thead>
<tr>
<th>Operands</th>
<th>Binary</th>
<th>Integer</th>
<th>Decimal</th>
<th>Number</th>
<th>Floating-point</th>
<th>Character</th>
<th>String</th>
<th>Graphic</th>
<th>String</th>
<th>Boolean</th>
<th>Date</th>
<th>Time</th>
<th>Time-stamp</th>
<th>Binary</th>
<th>String</th>
<th>UDT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary Integer</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>2</td>
</tr>
<tr>
<td>Decimal Number</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>2</td>
</tr>
<tr>
<td>Floating point</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>2</td>
</tr>
<tr>
<td>Decimal Floating point</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>2</td>
</tr>
<tr>
<td>Character String</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>2</td>
</tr>
<tr>
<td>Graphic String</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>2</td>
</tr>
<tr>
<td>Binary string</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>2</td>
</tr>
<tr>
<td>Date</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>2</td>
</tr>
<tr>
<td>Time</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>2</td>
</tr>
<tr>
<td>Time-stamp</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>2</td>
</tr>
<tr>
<td>Boolean</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>2</td>
</tr>
<tr>
<td>UDT</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>Yes</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

1 A TIMESTAMP value can be assigned to a TIME value; however, a TIME value cannot be assigned to a TIMESTAMP value and a TIMESTAMP value cannot be compared with a TIME value.

2 A user-defined distinct type value is only comparable to a value defined with the same user-defined distinct type. In general, assignments are supported between a distinct type value and its source data type. A user-defined structured type is not comparable and can only be assigned to an operand of the same structured type or one of its supertypes. Some additional assignment rules apply to row, cursor, and array types. For additional information see “User-defined type assignments” on page 121.

3 Support for assignment only (not comparison) and only for character strings defined as FOR BIT DATA.

4 For information on assignment and comparison of reference types, see “Reference type assignments” on page 123 and “Reference type comparisons” on page 129.

5 Only supported for Unicode databases.

6 Bit data and graphic strings are not compatible.

7 Variables of Boolean data type cannot be directly compared; comparison can only be done with the literal values TRUE, FALSE, NULL.
Assignments and comparisons

Numeric assignments

For numeric assignments, overflow is not allowed.

- When assigning to an exact numeric data type, overflow occurs if any digit of the whole part of the number would be eliminated. If necessary, the fractional part of a number is truncated.

- When assigning to an approximate numeric data type or decimal floating-point, overflow occurs if the most significant digit of the whole part of the number is eliminated. For floating-point and decimal floating-point numbers, the whole part of the number is the number that would result if the floating-point or decimal floating-point number were converted to a decimal number with unlimited precision. If necessary, rounding may cause the least significant digits of the number to be eliminated.

For decimal floating-point numbers, truncation of the whole part of the number is allowed and results in infinity with a warning.

For floating-point numbers, underflow is also not allowed. Underflow occurs for numbers between 1 and -1 if the most significant digit other than zero would be eliminated. For decimal floating-point, underflow is allowed and depending on the rounding mode, results in zero or the smallest positive number or the largest negative number that can be represented along with a warning.

An overflow or underflow warning is returned instead of an error if an overflow or underflow occurs on assignment to a host variable with an indicator variable. In this case, the number is not assigned to the host variable and the indicator variable is set to negative 2.

For decimal floating-point numbers, the CURRENT DECFLOAT Rounding MODE special register indicates the rounding mode in effect.

Assignments to integer

When a decimal, floating-point, or decimal floating-point number is assigned to an integer column or variable, the fractional part of the number is eliminated. As a result, a number between 1 and -1 is reduced to 0.

Assignments to decimal

When an integer is assigned to a decimal column or variable, the number is first converted to a temporary decimal number and then, if necessary, to the precision and scale of the target. The precision and scale of the temporary decimal number is 5,0 for a small integer, 11,0 for a large integer, or 19,0 for a big integer.

When a decimal number is assigned to a decimal column or variable, the number is converted, if necessary, to the precision and the scale of the target. The necessary number of leading zeros is added, and in the fractional part of the decimal number the necessary number of trailing zeros is added, or the necessary number of trailing digits is eliminated.

When a floating-point number is assigned to a decimal column or variable, the number is first converted to a temporary decimal number of precision 31, and then, if necessary, truncated to the precision and scale of the target. In this conversion, the number is rounded (using floating-point arithmetic) to a precision of 31 decimal digits. As a result, a number between 1 and -1 that is less than the smallest positive number or greater than the largest negative number that can be
Assignments and comparisons

represented in the decimal column or variable is reduced to 0. The scale is given
the largest possible value that allows the whole part of the number to be
represented without loss of significance.

When a decimal floating-point number is assigned to a decimal column or variable,
the number is rounded to the precision and scale of the decimal column or
variable. As a result, a number between 1 and -1 that is less than the smallest
positive number or greater than the largest negative number that can be
represented in the decimal column or variable is reduced to 0 or rounded to the
smallest positive or largest negative value that can be represented in the decimal
column or variable, depending on the rounding mode.

Assignments to floating-point

Floating-point numbers are approximations of real numbers. Hence, when an
integer, decimal, floating-point, or decimal floating-point number is assigned to a
floating-point column or variable, the result may not be identical to the original
number. The number is rounded to the precision of the floating-point column or
variable using floating-point arithmetic. A decimal floating-point value is first
converted to a string representation, and is then converted to a floating-point
number.

Assignments to decimal floating-point

When an integer number is assigned to a decimal floating-point column or
variable, the number is first converted to a temporary decimal number and then to
a decimal floating-point number. The precision and scale of the temporary decimal
number is 5,0 for a small integer, 11,0 for a large integer, or 19,0 for a big integer.
Rounding may occur when assigning a BIGINT to a DECFLOAT(16) column or
variable.

When a decimal number is assigned to a decimal floating-point column or
variable, the number is converted to the precision (16 or 34) of the target. Leading zeros are
eliminated. Depending on the precision and scale of the decimal number and the
precision of the target, the value might be rounded.

When a floating-point number is assigned to a decimal floating-point column or
variable, the number is first converted to a temporary string representation of the
floating-point number. The string representation of the number is then converted to
decimal floating-point.

When a DECFLOAT(16) number is assigned to a DECFLOAT(34) column or
variable, the resulting value is identical to the DECFLOAT(16) number.

When a DECFLOAT(34) number is assigned to a DECFLOAT(16) column or
variable, the exponent of the source is converted to the corresponding exponent in
the result format. The mantissa of the DECFLOAT(34) number is rounded to the
precision of the target.

Assignments from strings to numeric

When a string is assigned to a numeric data type, it is converted to the target
numeric data type using the rules for a CAST specification. For more information,
see “CAST specification” in the SQL Reference, Volume 1.
String assignments

There are two types of assignments:

- **In storage assignment**, a value is assigned and truncation of significant data is not desirable; for example, when assigning a value to a column.
- **In retrieval assignment**, a value is assigned and truncation is allowed; for example, when retrieving data from the database.

The rules for string assignment differ based on the assignment type.

Storage assignment

The basic rule is that the length of the string assigned to the target must not be greater than the length attribute of the target. If the length of the string is greater than the length attribute of the target, the following actions might occur:

- The string is assigned with trailing blanks truncated (from all string types except LOB strings) to fit the length attribute of the target.
- An error is returned (SQLSTATE 22001) when:
  - Non-blank characters would be truncated from other than a LOB string.
  - Any character (or byte) would be truncated from a LOB string.

If a string is assigned to a fixed-length target, and the length of the string is less than the length attribute of the target, the string is padded to the right with the necessary number of single-byte, double-byte, or UCS-2 blanks. The pad character is always a blank, even for columns defined with the FOR BIT DATA attribute. (UCS-2 defines several SPACE characters with different properties. For a Unicode database, the database manager always uses the ASCII SPACE at position x'0020' as UCS-2 blank. For an EUC database, the IDEOGRAPHIC SPACE at position x'3000' is used for padding GRAPHIC strings.)

Retrieval assignment

The length of a string that is assigned to a target can be longer than the length attribute of the target. When a string is assigned to a target, and the length of the string is longer than the length attribute of the target, the string is truncated on the right by the necessary number of characters (or bytes). When this occurs, a warning is returned (SQLSTATE 01004), and the value 'W' is assigned to the SQLWARN1 field of the SQLCA.

Furthermore, if an indicator variable is provided, and the source of the value is not a LOB, the indicator variable is set to the original length of the string.

If a character string is assigned to a fixed-length target, and the length of the string is less than the length attribute of the target, the string is padded to the right with the necessary number of single-byte, double-byte, or UCS-2 blanks. The pad character is always a blank, even for strings defined with the FOR BIT DATA attribute. (UCS-2 defines several SPACE characters with different properties. For a Unicode database, the database manager always uses the ASCII SPACE at position x'0020' as UCS-2 blank. For an EUC database, the IDEOGRAPHIC SPACE at position x'3000' is used for padding GRAPHIC strings.)

Retrieval assignment of C NUL-terminated host variables is handled on the basis of options that are specified with the PREP or BIND command.
Assignments and comparisons

Conversion rules for string assignments

A character string or graphic string assigned to a column, variable, or parameter is first converted, if necessary, to the code page of the target. Character conversion is necessary only if all of the following are true:

- The code pages are different.
- The string is neither null nor empty.
- Neither string has a code page value of 0 (FOR BIT DATA).

For Unicode databases, character strings can be assigned to a graphic column, and graphic strings can be assigned to a character column.

MBCS considerations for character string assignments

There are several considerations when assigning character strings that could contain both single and multi-byte characters. These considerations apply to all character strings, including those defined as FOR BIT DATA.

- Blank padding is always done using the single-byte blank character (X'20').
- Blank truncation is always done based on the single-byte blank character (X'20'). The double-byte blank character is treated like any other character with respect to truncation.
- Assignment of a character string to a host variable may result in fragmentation of MBCS characters if the target host variable is not large enough to contain the entire source string. If an MBCS character is fragmented, each byte of the MBCS character fragment in the target is set to a single-byte blank character (X'20'), no further bytes are moved from the source, and SQLWARN1 is set to 'W' to indicate truncation. Note that the same MBCS character fragment handling applies even when the character string is defined as FOR BIT DATA.

DBCS considerations for graphic string assignments

Graphic string assignments are processed in a manner analogous to that for character strings. For non-Unicode databases, graphic string data types are compatible only with other graphic string data types, and never with numeric, character string, or datetime data types. For Unicode databases, graphic string data types are compatible with character string data types. However, graphic and character string data types cannot be used interchangeably in the SELECT INTO or the VALUES INTO statement.

If a graphic string value is assigned to a graphic string column, the length of the value must not be greater than the length of the column.

If a graphic string value (the ‘source’ string) is assigned to a fixed length graphic string data type (the ‘target’, which can be a column, variable, or parameter), and the length of the source string is less than that of the target, the target will contain a copy of the source string which has been padded on the right with the necessary number of double-byte blank characters to create a value whose length equals that of the target.

If a graphic string value is assigned to a graphic string host variable and the length of the source string is greater than the length of the host variable, the host variable will contain a copy of the source string which has been truncated on the right by the necessary number of double-byte characters to create a value whose length equals that of the host variable. (Note that for this scenario, truncation need not be
Assignments and comparisons

cconcerned with bisection of a double-byte character; if bisection were to occur, either the source value or target host variable would be an ill-defined graphic string data type.) The warning flag SQLWARN1 in the SQLCA will be set to ‘W’. The indicator variable, if specified, will contain the original length (in double-byte characters) of the source string. In the case of DBCLOB, however, the indicator variable does not contain the original length.

Retrieval assignment of C NUL-terminated host variables (declared using wchar_t) is handled based on options specified with the PREP or BIND command.

Assignments from numeric to strings

When a number is assigned to a string data type, it is converted to the target string data type using the rules for a CAST specification. For more information, see “CAST specification” in the SQL Reference, Volume 1.

If a nonblank character is truncated during the cast of a numeric value to a character or graphic data type, a warning is returned. This truncation behavior is unlike the assignment to a character or graphic data type that follows storage assignment rules, where if a nonblank character is truncated during assignment, an error is returned.

Datetime assignments

A TIME value can be assigned only to a TIME column or to a string variable or string column.

A DATE can be assigned to a DATE, TIMESTAMP or string data type. When a DATE value is assigned to a TIMESTAMP data type, the missing time information is assumed to be all zeros.

A TIMESTAMP value can be assigned to a DATE, TIME, TIMESTAMP or string data type. When a TIMESTAMP value is assigned to a DATE data type, the date portion is extracted and the time portion is truncated. When a TIMESTAMP value is assigned to a TIME data type, the date portion is ignored and the time portion is extracted, but with the fractional seconds truncated. When a TIMESTAMP value is assigned to a TIMESTAMP with lower precision, the excess fractional seconds are truncated. When a TIMESTAMP value is assigned to a TIMESTAMP with higher precision, missing digits are assumed to be zeros.

The assignment must not be to a CLOB, DBCLOB, or BLOB variable or column.

When a datetime value is assigned to a string variable or string column, conversion to a string representation is automatic. Leading zeros are not omitted from any part of the date, time, or timestamp. The required length of the target will vary, depending on the format of the string representation. If the length of the target is greater than required, and the target is a fixed-length string, it is padded on the right with blanks. If the length of the target is less than required, the result depends on the type of datetime value involved, and on the type of target.

When the target is not a host variable and has a character data type, truncation is not allowed. The length attribute of the column must be at least 10 for a date, 8 for a time, 19 for a TIMESTAMP(0), and 20+p for TIMESTAMP(p).

When the target is a string host variable, the following rules apply:
Assignments and comparisons

- **For a DATE:** If the length of the host variable is less than 10 characters, an error is returned.
- **For a TIME:** If the USA format is used, the length of the host variable must not be less than 8 characters; in other formats the length must not be less than 5 characters.

  If ISO or JIS formats are used, and if the length of the host variable is less than 8 characters, the seconds part of the time is omitted from the result and assigned to the indicator variable, if provided. The SQLWARN1 field of the SQLCA is set to indicate the omission.
- **For a TIMESTAMP:** If the length of the host variable is less than 19 characters, an error is returned. If the length is less than 32 characters, but greater than or equal to 19 characters, trailing digits of the fractional seconds part of the value are omitted. The SQLWARN1 field of the SQLCA is set to indicate the omission.

When a DATE is assigned to a TIMESTAMP, the time and fractional components of the timestamp are set to midnight and 0, respectively. When a TIMESTAMP is assigned to a DATE, the date portion is extracted and the time and fractional components are truncated.

When a TIMESTAMP is assigned to a TIME, the DATE portion is ignored and the fractional components are truncated.

XML assignments

The general rule for XML assignments is that only an XML value can be assigned to XML columns or to XML variables. There are exceptions to this rule, as follows.

- **Processing of input XML host variables:** This is a special case of the XML assignment rule, because the host variable is based on a string value. To make the assignment to XML within SQL, the string value is implicitly parsed into an XML value using the setting of the CURRENT IMPLICIT XMLPARSE OPTION special register. This determines whether to preserve or to strip whitespace, unless the host variable is an argument of the XMLVALIDATE function, which always strips unnecessary whitespace.

- **Assigning strings to input parameter markers of data type XML:** If an input parameter marker has an implicit or explicit data type of XML, the value bound (assigned) to that parameter marker could be a character string variable, graphic string variable, or binary string variable. In this case, the string value is implicitly parsed into an XML value using the setting of the CURRENT IMPLICIT XMLPARSE OPTION special register to determine whether to preserve or to strip whitespace, unless the parameter marker is an argument of the XMLVALIDATE function, which always strips unnecessary whitespace.

- **Assigning strings directly to XML columns in data change statements:** If assigning directly to a column of type XML in a data change statement, the assigned expression can also be a character string or a binary string. In this case, the result of XMLPARSE (DOCUMENT expression STRIP WHITESPACE) is assigned to the target column. The supported string data types are defined by the supported arguments for the XMLPARSE function. Note that this XML assignment exception does not allow character or binary string values to be assigned to SQL variables or to SQL parameters of data type XML.

- **Assigning XML to strings on retrieval:** If retrieving XML values into host variables using a FETCH INTO statement or an EXECUTE INTO statement in embedded SQL, the data type of the host variable can be CLOB, DBCLOB, or BLOB. If using other application programming interfaces (such as CLI, JDBC, or .NET), XML values can be retrieved into the character, graphic, or binary string
types that are supported by the application programming interface. In all of these cases, the XML value is implicitly serialized to a string encoded in UTF-8 and, for character or graphic string variables, converted into the client code page.

Character string or binary string values cannot be retrieved into XML host variables. Values in XML host variables cannot be assigned to columns, SQL variables, or SQL parameters of a character string data type or a binary string data type.

Assignment to XML parameters and variables in inlined SQL bodied UDFs and SQL procedures is done by reference. Passing parameters of data type XML to invoke an inlined SQL UDF or SQL procedure is also done by reference. When XML values are passed by reference, any input node trees are used directly. This direct usage preserves all properties, including document order, the original node identities, and all parent properties.

**User-defined type assignments**

For distinct types and structured types, different rules are applied for assignments to host variables than are used for all other assignments.

Distinct Types: Assignment to host variables is done based on the source type of the distinct type. That is, it follows the rule:

- A value of a distinct type on the right hand side of an assignment is assignable to a host variable on the left hand side if and only if the source type of this distinct type is assignable to this host variable.

If the target of the assignment is a column based on a distinct type, the source data type must be castable to the target data type.

Structured Types: Assignment to and from host variables is based on the declared type of the host variable; that is, it follows the rule:

- A value of a structured type on the right hand side of an assignment is assignable to a host variable on the left hand side if and only if the declared type of the host variable is the structured type or a supertype of the structured type.

If the target of the assignment is a column of a structured type, the source data type must be the target data type or a subtype of the target data type.

For array types, different rules are applied for assignments to SQL variables and parameters. The validity of an assignment to an SQL variable or parameter is determined according to the following rules:

- If the right hand side of the assignment is an SQL variable or parameter, an invocation of the TRIM_ARRAY function, an invocation of the ARRAY_DELETE function, or a CAST expression, then its type must be the same as the type of the SQL variable or parameter on the left hand side of the assignment.
- If the right hand side of the assignment is an array constructor or an invocation of the ARRAY_AGG function, then it is implicitly cast to the type of the SQL variable or parameter on the left hand side.

For example, assuming that the type of variable \( V \) is MYARRAY, the statement:

\[
\text{SET } V = \text{ARRAY}[1,2,3];
\]
Assignments and comparisons

is equivalent to:

\[
\text{SET } V = \text{CAST(ARRAY}[1,2,3] \text{ AS MYARRAY);} \\
\]

And the statement:

\[
\text{SELECT ARRAY_AGG(C1) INTO V FROM T} \\
\]

is equivalent to:

\[
\text{SELECT CAST(ARRAY_AGG(C1) AS MYARRAY) INTO V FROM T} \\
\]

Additional information about specific user-defined types is in the sections that follow:

Array type assignments

The value for an element of an array must be assignable to the data type of the array elements. The assignment rules for that data type apply to the value assignment. The value specified for an index in the array must be assignable to the data type of the index for the array. The assignment rules for that data type apply to the value assignment. For an ordinary array the index data type is INTEGER and for an associative array the data type is either INTEGER or VARCHAR(n), where n is any valid length attribute for the VARCHAR data type. If the index value for an assignment to an ordinary array is larger than the current cardinality of the array, then the cardinality of the array is increased to the new index value, provided the value does not exceed the maximum value for an INTEGER data type. An assignment of one new element to an associative array increases the cardinality by exactly 1 since the index values can be sparse.

The following are valid assignments involving array type values:

- Array variable to another array variable with the same array type as the source variable.
- An expression of type array to an array variable, where the array element type in the source expression is assignable to the array element type in the target array variable.

Row type assignments

Assignments to fields within a row variable must conform to the same rules as if the field itself was a variable of the same data type as the field. A row variable can be assigned only to a row variable with the same user-defined row type. When using FETCH, SELECT, or VALUES INTO to assign values to a row variable, the source value types must be assignable to the target row fields. If the source or the target variable (or both) of an assignment is anchored to the row of a table or view, the number of fields must be the same and the field types of the source value must be assignable to the field types of the target value.

Cursor type assignments

Assignments to cursors depend on the type of cursor. The following values are assignable to a variable or parameter of built-in type CURSOR:

- A cursor value constructor
- A value of built-in type CURSOR
- A value of any user-defined cursor type
Assignments and comparisons

The following values are assignable to a variable or parameter of a weakly-typed user-defined cursor type:

- A cursor value constructor
- A value of built-in type CURSOR
- A value of a user-defined cursor type with the same type name

The following values are assignable to a variable or parameter of strongly-typed user-defined cursor type:

- A cursor value constructor
- A value of a user-defined cursor type with the same type name

Boolean type assignments

The following system-defined values are assignable to a variable, parameter, or return type of built-in type BOOLEAN:

- TRUE
- FALSE
- NULL

The result of the evaluation of a search condition can also be assigned. If the search condition evaluates to unknown, the value of NULL is assigned.

Reference type assignments

A reference type with a target type of \( T \) can be assigned to a reference type column that is also a reference type with target type of \( S \) where \( S \) is a supertype of \( T \). If an assignment is made to a scoped reference column or variable, no check is performed to ensure that the actual value being assigned exists in the target table or view defined by the scope.

Assignment to host variables is done based on the representation type of the reference type. That is, it follows the rule:

- A value of a reference type on the right hand side of an assignment is assignable to a host variable on the left hand side if and only if the representation type of this reference type is assignable to this host variable.

If the target of the assignment is a column, and the right hand side of the assignment is a host variable, the host variable must be explicitly cast to the reference type of the target column.

Numeric comparisons

Numbers are compared algebraically; that is, with regard to sign. For example, -2 is less than +1.

If one number is an integer and the other is decimal, the comparison is made with a temporary copy of the integer, which has been converted to decimal.

When decimal numbers with different scales are compared, the comparison is made with a temporary copy of one of the numbers that has been extended with trailing zeros so that its fractional part has the same number of digits as the other number.
Assignments and comparisons

If one number is floating-point and the other is integer or decimal, the comparison is made with a temporary copy of the other number, which has been converted to double-precision floating-point.

Two floating-point numbers are equal only if the bit configurations of their normalized forms are identical.

If one number is decimal floating-point and the other number is integer, decimal, single precision floating-point, or double precision floating-point, the comparison is made with a temporary copy of the other number, which has been converted to decimal floating-point.

If one number is DECFLOAT(16) and the other number is DECFLOAT(34), the DECFLOAT(16) value is converted to DECFLOAT(34) before the comparison is made.

The decimal floating-point data type supports both positive and negative zero. Positive and negative zero have different binary representations, but the = (equal) predicate will return true for comparisons of negative and positive zero.

The COMPARE_DECFLOAT and TOTALORDER scalar functions can be used to perform comparisons at a binary level if, for example, a comparison of 2.0 <> 2.00 is required.

The decimal floating-point data type supports the specification of negative and positive NaN (quiet and signalling), and negative and positive infinity. From an SQL perspective, INFINITY = INFINITY, NaN = NaN, SNAN = SNAN, and -0 = 0.

The comparison and ordering rules for special values are as follows:

• (+/-) INFINITY compares equal only to (+/-) INFINITY of the same sign.
• (+/-) NaN compares equal only to (+/-) NaN of the same sign.
• (+/-) SNAN compares equal only to (+/-) SNAN of the same sign.

The ordering among different special values is as follows:

• -NAN < -SNAN < -INFINITY < 0 < INFINITY < SNAN < NAN

When string and numeric data types are compared, the string is cast to DECFLOAT(34) using the rules for a CAST specification. For more information, see “CAST specification” in the SQL Reference, Volume 1. The string must contain a valid string representation of a number.

String comparisons

Character strings are compared according to the collating sequence specified when the database was created, except those with a FOR BIT DATA attribute, which are always compared according to their bit values.

When comparing character strings of unequal lengths, the comparison is made using a logical copy of the shorter string, which is padded on the right with blanks sufficient to extend its length to that of the longer string. This logical extension is done for all character strings, including those tagged as FOR BIT DATA.

Character strings (except character strings tagged as FOR BIT DATA) are compared according to the collating sequence specified when the database was created. For
example, the default collating sequence supplied by the database manager may
give lowercase and uppercase versions of the same character the same weight. The
database manager performs a two-pass comparison to ensure that only identical
strings are considered equal to each other. In the first pass, strings are compared
according to the database collating sequence. If the weights of the characters in the
strings are equal, a second "tie-breaker" pass is performed to compare the strings
on the basis of their actual code point values.

Two strings are equal if they are both empty or if all corresponding bytes are
equal. If either operand is null, the result is unknown.

LOB strings are not supported in any comparison operations that use the basic
comparison operators (=, <>, <, >, <=, and >=). They are supported in comparisons
using the LIKE predicate and the POSSTR function.

Portions of strings can be compared using the SUBSTR and VARCHAR scalar
functions. For example, given the columns:

| MY_SHORT_CLOB  | CLOB(300)          |
| MY_LONG_VAR    | VARCHAR(8000)      |

then the following is valid:

WHERE VARCHAR(MY_SHORT_CLOB) > VARCHAR(SUBSTR(MY_LONG_VAR,1,300))

Examples:

For these examples, 'A', 'Á', 'á', and 'á', have the code point values X'41', X'C1',
X'61', and X'E1' respectively.

Consider a collating sequence where the characters 'A', 'Á', 'á', 'á' have weights
136, 139, 135, and 138. Then the characters sort in the order of their weights as
follows:

'a' < 'A' < 'á' < 'Á'

Now consider four DBCS characters D1, D2, D3, and D4 with code points 0xC141,
0xC161, 0xE141, and 0xE161, respectively. If these DBCS characters are in CHAR
columns, they sort as a sequence of bytes according to the collation weights of
those bytes. First bytes have weights of 138 and 139, therefore D3 and D4 come
before D2 and D1; second bytes have weights of 135 and 136. Hence, the order is
as follows:

D4 < D3 < D2 < D1

However, if the values being compared have the FOR BIT DATA attribute, or if
these DBCS characters were stored in a GRAPHIC column, the collation weights
are ignored, and characters are compared according to their code points as follows:

'A' < 'a' < 'Á' < 'á'

The DBCS characters sort as sequence of bytes, in the order of code points as
follows:

D1 < D2 < D3 < D4

Now consider a collating sequence where the characters 'A', 'Á', 'á', 'á' have
(non-unique) weights 74, 75, 74, and 75. Considering collation weights alone (first
pass), 'a' is equal to 'A', and 'á' is equal to 'Á'. The code points of the characters
are used to break the tie (second pass) as follows:

'A' < 'a' < 'Á' < 'á'
Assignments and comparisons

DBCS characters in CHAR columns sort a sequence of bytes, according to their weights (first pass) and then according to their code points to break the tie (second pass). First bytes have equal weights, so the code points (0xC1 and 0xE1) break the tie. Therefore, characters D1 and D2 sort before characters D3 and D4. Then the second bytes are compared in similar way, and the final result is as follows:

\[ D_1 < D_2 < D_3 < D_4 \]

Once again, if the data in CHAR columns have the FOR BIT DATA attribute, or if the DBCS characters are stored in a GRAPHIC column, the collation weights are ignored, and characters are compared according to their code points:

\[ D_1 < D_2 < D_3 < D_4 \]

For this particular example, the result happens to be the same as when collation weights were used, but obviously this is not always the case.

Conversion rules for comparison

When two strings are compared, one of the strings is first converted, if necessary, to the encoding scheme and code page of the other string.

Ordering of results

Results that require sorting are ordered based on the string comparison rules discussed in "String comparisons" on page 124. The comparison is performed at the database server. On returning results to the client application, code page conversion may be performed. This subsequent code page conversion does not affect the order of the server-determined result set.

MBCS considerations for string comparisons

Mixed SBCS/MBCS character strings are compared according to the collating sequence specified when the database was created. For databases created with default (SYSTEM) collation sequence, all single-byte ASCII characters are sorted in correct order, but double-byte characters are not necessarily in code point sequence. For databases created with IDENTITY sequence, all double-byte characters are correctly sorted in their code point order, but single-byte ASCII characters are sorted in their code point order as well. For databases created with COMPATIBILITY sequence, a compromise order is used that sorts properly for most double-byte characters, and is almost correct for ASCII. This was the default collation table in DB2 Version 2.

Mixed character strings are compared byte-by-byte. This may result in unusual results for multi-byte characters that occur in mixed strings, because each byte is considered independently.

Example:

For this example, 'A', 'B', 'a', and 'b' double-byte characters have the code point values X'8260', X'8261', X'8281', and X'8282', respectively.

Consider a collating sequence where the code points X'8260', X'8261', X'8281', and X'8282' have weights 96, 65, 193, and 194. Then:

\[ 'B' < 'A' < 'a' < 'b' \]

and
Graphic string comparisons are processed in a manner analogous to that for character strings.

Graphic string comparisons are valid between all graphic string data types except DBCLOB.

For graphic strings, the collating sequence of the database is not used. Instead, graphic strings are always compared based on the numeric (binary) values of their corresponding bytes.

Using the previous example, if the literals were graphic strings, then:

'A' < 'B' < 'a' < 'b'

and

'AA' < 'AB' < 'Aa' < 'Ab' < 'aA' < 'aB' < 'aa' < 'ab'

When comparing graphic strings of unequal lengths, the comparison is made using a logical copy of the shorter string which is padded on the right with double-byte blank characters sufficient to extend its length to that of the longer string.

Two graphic values are equal if they are both empty or if all corresponding graphics are equal. If either operand is null, the result is unknown. If two values are not equal, their relation is determined by a simple binary string comparison.

As indicated in this section, comparing strings on a byte by byte basis can produce unusual results; that is, a result that differs from what would be expected in a character by character comparison. The examples shown here assume the same MBCS code page, however, the situation can be further complicated when using different multi-byte code pages with the same national language. For example, consider the case of comparing a string from a Japanese DBCS code page and a Japanese EUC code page.

**Datetime comparisons**

A DATE, TIME, or TIMESTAMP value may be compared either with another value of the same data type or with a string representation of that data type. A DATE or a string representation of a date can also be compared with a TIMESTAMP, where the missing time information for the date value is assumed to be all zeros. All comparisons are chronological, which means the farther a point in time is from January 1, 0001, the greater the value of that point in time.

Comparisons involving TIME values and string representations of time values always include seconds. If the string representation omits seconds, zero seconds is implied.

Comparisons involving TIMESTAMP values are chronological without regard to representations that might be considered equivalent.

Example:

```
TIMESTAMP('1990-02-23-00.00.00') > '1990-02-22-24.00.00'
```
Assignments and comparisons

User-defined type comparisons

Values with a user-defined distinct type can only be compared with values of exactly the same user-defined distinct type. The user-defined distinct type must have been defined using the WITH COMPARISONS clause.

Example:

Given the following YOUTH distinct type and CAMP_DB2_ROSTER table:

```
CREATE TYPE YOUTH AS INTEGER WITH COMPARISONS

CREATE TABLE CAMP_DB2_ROSTER
    ( NAME VARCHAR(20),
      ATTENDEE_NUMBER INTEGER NOT NULL,
      AGE YOUTH,
      HIGH_SCHOOL_LEVEL YOUTH)
```

The following comparison is valid:

```
SELECT * FROM CAMP_DB2_ROSTER
WHERE AGE > HIGH_SCHOOL_LEVEL
```

The following comparison is not valid:

```
SELECT * FROM CAMP_DB2_ROSTER
WHERE AGE > ATTENDEE_NUMBER
```

However, AGE can be compared to ATTENDEE_NUMBER by using a function or CAST specification to cast between the distinct type and the source type. The following comparisons are all valid:

```
SELECT * FROM CAMP_DB2_ROSTER
WHERE INTEGER(AGE) > ATTENDEE_NUMBER

SELECT * FROM CAMP_DB2_ROSTER
WHERE CAST(AGE AS INTEGER) > ATTENDEE_NUMBER

SELECT * FROM CAMP_DB2_ROSTER
WHERE AGE > YOUTH(ATTENDEE_NUMBER)

SELECT * FROM CAMP_DB2_ROSTER
WHERE AGE > CAST(ATTENDEE_NUMBER AS YOUTH)
```

Values with a user-defined structured type cannot be compared with any other value (the NULL predicate and the TYPE predicate can be used).

Comparisons of array type values are not supported. Elements of arrays can be compared based on the comparison rules for the data type of the elements.

Row type comparisons

A row variable cannot be compared to another row variable even if the row type name is the same. Individual fields within a row type can be compared to other values and the comparison rules for the data type of the field apply.

Cursor type comparisons

A cursor variable cannot be compared to another cursor variable even if the cursor type name is the same.
Assignments and comparisons

Boolean type comparisons

A Boolean value can be compared with a Boolean literal value. A value of TRUE is greater than a value of FALSE.

Reference type comparisons

Reference type values can be compared only if their target types have a common supertype. The appropriate comparison function will only be found if the schema name of the common supertype is included in the SQL path. The comparison is performed using the representation type of the reference types. The scope of the reference is not considered in the comparison.

XML comparisons in a non-Unicode database

When performed in a non-Unicode database, comparisons between XML data and character or graphic string values require a code page conversion of one of the two sets of data being compared. Character or graphic values used in an SQL or XQuery statement, either as a query predicate or as a host variable with a character or graphic string data type, are converted to the database code page prior to comparison. If any characters included in this data have code points that are not part of the database code page, substitution characters are added in their place, potentially causing unexpected results for the query.

For example, a client with a UTF-8 code page is used to connect to a database server created with the Greek encoding ISO8859-7. The expression \(\Sigma \Gamma\) is sent as the predicate of an XQuery statement, where \(\Sigma\) represents the Greek sigma character in Unicode (U+03A3) and \(\Gamma\) represents the mathematical symbol sigma in Unicode (U+2211). This expression is first converted to the database code page, so that both \(\Sigma\) characters are converted to the equivalent code point for sigma in the Greek database code page, 0xD3. We may denote this code point as \(\Sigma_A\). The newly converted expression \(\Sigma_A \Sigma_A\) is then converted again to UTF-8 for comparison with the target XML data. Since the distinction between these two code points was lost as a result of the code page conversion required to pass the predicate expression into the database, the two initially distinct values \(\Sigma\) and \(\Gamma\) are passed to the XML parser as the expression \(\Sigma \Sigma\). This expression then fails to match when compared to the value \(\Sigma \Sigma\) in an XML document.

One way to avoid the unexpected query results that may be caused by code page conversion issues is to ensure that all characters used in a query expression have matching code points in the database code page. Characters that do not have matching code points can be included through the use of a Unicode character entity reference. A character entity reference will always bypass code page conversion. For example, using the character entity reference &#2211; in place of the \(\Sigma\) character ensures that the correct Unicode code point is used for the comparison, regardless of the database code page.
Rules for result data types

The data types of a result are determined by rules which are applied to the operands in an operation. This section explains those rules.

These rules apply to:

- Corresponding columns in fullselects of set operations (UNION, INTERSECT and EXCEPT)
- Result expressions of a CASE expression and the DECODE scalar function
- Arguments of the scalar function COALESCE (also NVL and VALUE)
- Arguments of the scalar functions GREATEST, LEAST, MAX, and MIN
- Expression values of the in list of an IN predicate
- Corresponding expressions of a multiple row VALUES clause
- Expression values for the elements in an array constructor
- Arguments of a BETWEEN predicate (except if the data types of all operands are numeric)
- Arguments for the aggregation group ranges in OLAP specifications

These rules are applied subject to other restrictions on long strings for the various operations.

The rules involving various data types follow. In some cases, a table is used to show the possible result data types. The LONG VARCHAR and LONG VARGRAPHIC data types continue to be supported but are deprecated and not recommended.

These tables identify the data type of the result, including the applicable length or precision and scale. The result type is determined by considering the operands. If there is more than one pair of operands, start by considering the first pair. This gives a result type which is considered with the next operand to determine the next result type, and so on. The last intermediate result type and the last operand determine the result type for the operation. Processing of operations is done from left to right so that the intermediate result types are important when operations are repeated. For example, consider a situation involving:

```
CHAR(2) UNION CHAR(4) UNION VARCHAR(3)
```

The first pair results in a type of CHAR(4). The result values always have 4 bytes. The final result type is VARCHAR(4). Values in the result from the first UNION operation will always have a length of 4.

**Character strings**

A character string value is compatible with another character string value. Character strings include data types CHAR, VARCHAR, and CLOB.

<table>
<thead>
<tr>
<th>If one operand is...</th>
<th>And the other operand is...</th>
<th>The data type of the result is...</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR(x)</td>
<td>CHAR(y)</td>
<td>CHAR(z) where z = max(x,y)</td>
</tr>
<tr>
<td>CHAR(x)</td>
<td>VARCHAR(y)</td>
<td>VARCHAR(z) where z = max(x,y)</td>
</tr>
<tr>
<td>VARCHAR(x)</td>
<td>CHAR(y)</td>
<td>VARCHAR(z) where z = max(x,y)</td>
</tr>
<tr>
<td></td>
<td>or</td>
<td></td>
</tr>
<tr>
<td></td>
<td>VARCHAR(y)</td>
<td></td>
</tr>
</tbody>
</table>
Rules for result data types

<table>
<thead>
<tr>
<th>If one operand is...</th>
<th>And the other operand is...</th>
<th>The data type of the result is...</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLOB(x)</td>
<td>CHAR(y), VARCHAR(y), or CLOB(y)</td>
<td>CLOB(z) where ( z = \max(x,y) )</td>
</tr>
</tbody>
</table>

The code page of the result character string will be derived based on the rules for string conversions.

**Graphic strings**

A graphic string value is compatible with another graphic string value. Graphic strings include data types GRAPHIC, VARGRAPHIC, and DBCLOB.

<table>
<thead>
<tr>
<th>If one operand is...</th>
<th>And the other operand is...</th>
<th>The data type of the result is...</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRAPHIC(x)</td>
<td>GRAPHIC(y)</td>
<td>GRAPHIC(z) where ( z = \max(x,y) )</td>
</tr>
<tr>
<td>VARGRAPHIC(x)</td>
<td>GRAPHIC(y) OR VARGRAPHIC(y)</td>
<td>VARGRAPHIC(z) where ( z = \max(x,y) )</td>
</tr>
<tr>
<td>DBCLOB(x)</td>
<td>GRAPHIC(y), VARGRAPHIC(y), or DBCLOB(y)</td>
<td>DBCLOB(z) where ( z = \max(x,y) )</td>
</tr>
</tbody>
</table>

The code page of the result graphic string will be derived based on the rules for string conversions.

**Character and graphic strings in a Unicode database**

In a Unicode database, a character string value is compatible with a graphic string value.

<table>
<thead>
<tr>
<th>If one operand is...</th>
<th>And the other operand is...</th>
<th>The data type of the result is...</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRAPHIC(x)</td>
<td>CHAR(y) or GRAPHIC(y)</td>
<td>GRAPHIC(z) where ( z = \max(x,y) )</td>
</tr>
<tr>
<td>VARGRAPHIC(x)</td>
<td>CHAR(y) or VARCHAR(y)</td>
<td>VARGRAPHIC(z) where ( z = \max(x,y) )</td>
</tr>
<tr>
<td>VARCHAR(x)</td>
<td>GRAPHIC(y) or VARGRAPHIC</td>
<td>VARGRAPHIC(z) where ( z = \max(x,y) )</td>
</tr>
<tr>
<td>DBCLOB(x)</td>
<td>CHAR(y) or VARCHAR(y) or CLOB(y)</td>
<td>DBCLOB(z) where ( z = \max(x,y) )</td>
</tr>
<tr>
<td>CLOB(x)</td>
<td>GRAPHIC(y) or VARGRAPHIC(y)</td>
<td>DBCLOB(z) where ( z = \max(x,y) )</td>
</tr>
</tbody>
</table>

**Binary large object (BLOB)**

A binary string (BLOB) value is compatible only with another binary string (BLOB) value. The BLOB scalar function can be used to cast from other types if they should be treated as BLOB types. The length of the result BLOB is the largest length of all the data types.
## Rules for result data types

### Numeric

Numeric types are compatible with other numeric data types, character-string data types (except CLOB), and in a Unicode database, graphic-string data types (except DBCLOB). Numeric types include SMALLINT, INTEGER, BIGINT, DECIMAL, REAL, DOUBLE, and DECFLOAT.

<table>
<thead>
<tr>
<th>If one operand is...</th>
<th>And the other operand is...</th>
<th>The data type of the result is...</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMALLINT</td>
<td>SMALLINT</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>String</td>
<td>DECFLOAT(34)</td>
</tr>
<tr>
<td>INTEGER</td>
<td>SMALLINT or INTEGER</td>
<td>INTEGER</td>
</tr>
<tr>
<td>INTEGER</td>
<td>String</td>
<td>DECFLOAT(34)</td>
</tr>
<tr>
<td>BIGINT</td>
<td>SMALLINT, INTEGER, or BIGINT</td>
<td>BIGINT</td>
</tr>
<tr>
<td>BIGINT</td>
<td>String</td>
<td>DECFLOAT(34)</td>
</tr>
<tr>
<td>DECIMAL(w,x)</td>
<td>SMALLINT</td>
<td>DECIMAL(p,x) where p = x+max(w-x,5) (^1)</td>
</tr>
<tr>
<td>DECIMAL(w,x)</td>
<td>INTEGER</td>
<td>DECIMAL(p,x) where p = x+max(w-x,11) (^1)</td>
</tr>
<tr>
<td>DECIMAL(w,x)</td>
<td>BIGINT</td>
<td>DECIMAL(p,x) where p = x+max(w-x,19) (^1)</td>
</tr>
<tr>
<td>DECIMAL(w,x)</td>
<td>DECIMAL(y,z)</td>
<td>DECIMAL(p,s) where p = max(x,z)+max(w-x,y-z)(^1) = max(x,z)</td>
</tr>
<tr>
<td>DECIMAL(w,x)</td>
<td>String</td>
<td>DECFLOAT(34)</td>
</tr>
<tr>
<td>REAL</td>
<td>REAL</td>
<td>REAL</td>
</tr>
<tr>
<td>REAL</td>
<td>SMALLINT, INTEGER, BIGINT, or DECIMAL</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>REAL</td>
<td>String</td>
<td>DECFLOAT(34)</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>SMALLINT, INTEGER, BIGINT, DECIMAL, REAL, or DOUBLE</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>String</td>
<td>DECFLOAT(34)</td>
</tr>
<tr>
<td>DECFLOAT(n)</td>
<td>SMALLINT, INTEGER, DECIMAL (&lt;=16,s), REAL, or DOUBLE</td>
<td>DECFLOAT(n)</td>
</tr>
<tr>
<td>DECFLOAT(n)</td>
<td>BIGINT or DECIMAL (&gt;16,s)</td>
<td>DECFLOAT(34)</td>
</tr>
<tr>
<td>DECFLOAT(n)</td>
<td>DECFLOAT(m)</td>
<td>DECFLOAT(MAX(n,m))</td>
</tr>
<tr>
<td>DECFLOAT(n)</td>
<td>String</td>
<td>DECFLOAT(34)</td>
</tr>
</tbody>
</table>

\(^1\) Precision cannot exceed 31.

### Datetime

Datetime data types are compatible with other operands of the same data type or any CHAR or VARCHAR expression that contains a valid string representation of the same data type. In addition, DATE is compatible with TIMESTAMP and the other operand of a TIMESTAMP can be the string representation of a timestamp or
Rules for result data types

A date. In a Unicode database, character and graphic strings are compatible which implies that GRAPHIC or VARGRAPHIC string representations of datetime values are compatible with other datetime operands.

Table 16. Result data types with datetime operands

<table>
<thead>
<tr>
<th>If one operand is...</th>
<th>And the other operand is...</th>
<th>The data type of the result is...</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATE</td>
<td>DATE, CHAR(y), or VARCHAR(y)</td>
<td>DATE</td>
</tr>
<tr>
<td>TIME</td>
<td>TIME, CHAR(y), or VARCHAR(y)</td>
<td>TIME</td>
</tr>
<tr>
<td>TIMESTAMP(x)</td>
<td>TIMESTAMP(y)</td>
<td>TIMESTAMP(max(x,y))</td>
</tr>
<tr>
<td>TIMESTAMP(x)</td>
<td>DATE, CHAR(y), or VARCHAR(y)</td>
<td>TIMESTAMP(x)</td>
</tr>
</tbody>
</table>

XML

An XML value is compatible with another XML value. The data type of the result is XML.

Boolean

A Boolean value is compatible with another Boolean value. The data type of the result is BOOLEAN.

Distinct types

A user-defined distinct type value is compatible only with another value of the same user-defined distinct type. The data type of the result is the user-defined distinct type.

Array data types

A user-defined array data type value is compatible only with another value of the same user-defined array data type. The data type of the result is the user-defined array data type.

Cursor data types

A CURSOR value is compatible with another CURSOR value. The result data type is CURSOR. A user-defined cursor data type value is compatible only with another value of the same user-defined cursor data type. The data type of the result is the user-defined cursor data type.

Row data types

A user-defined row data type value is compatible only with another value of the same user-defined row data type. The data type of the result is the user-defined row data type.

Reference types

A reference type value is compatible with another value of the same reference type provided that their target types have a common supertype. The data type of the result is a reference type having the common supertype as the target type. If all operands have the identical scope table, the result has that scope table. Otherwise the result is unscoped.
Rules for result data types

**Structured types**

A structured type value is compatible with another value of the same structured type provided that they have a common supertype. The static data type of the resulting structured type column is the structured type that is the least common supertype of either column.

For example, consider the following structured type hierarchy,

```
 A
 / \ 
 B   C
 / \ 
 D   E
 / \ 
 F   G
```

Structured types of the static type E and F are compatible with the resulting static type of B, which is the least common super type of E and F.

**Nullable attribute of result**

With the exception of INTERSECT and EXCEPT, the result allows nulls unless both operands do not allow nulls.

- For INTERSECT, if either operand does not allow nulls the result does not allow nulls (the intersection would never be null).
- For EXCEPT, if the first operand does not allow nulls the result does not allow nulls (the result can only be values from the first operand).
Rules for string conversions

The code page used to perform an operation is determined by rules which are applied to the operands in that operation. This section explains those rules.

These rules apply to:
- Corresponding string columns in fullselects with set operations (UNION, INTERSECT and EXCEPT)
- Operands of concatenation
- Operands of predicates (with the exception of LIKE)
- Result expressions of a CASE expression and the DECODE scalar function
- Arguments of the scalar function COALESCE (also NVL and VALUE)
- Arguments of the scalar functions GREATEST, LEAST, MAX, and MIN
- The source-string and insert-string arguments of the scalar function OVERLAY (and INSERT)
- Expression values of the in list of an IN predicate
- Corresponding expressions of a multiple row VALUES clause.

In each case, the code page of the result is determined at bind time, and the execution of the operation may involve conversion of strings to the code page identified by that code page. A character that has no valid conversion is mapped to the substitution character for the character set and SQLWARN10 is set to 'W' in the SQLCA.

The code page of the result is determined by the code pages of the operands. The code pages of the first two operands determine an intermediate result code page, this code page and the code page of the next operand determine a new intermediate result code page (if applicable), and so on. The last intermediate result code page and the code page of the last operand determine the code page of the result string or column. For each pair of code pages, the result is determined by the sequential application of the following rules:
- If the code pages are equal, the result is that code page.
- If either code page is BIT DATA (code page 0), the result code page is BIT DATA.
- In a Unicode database, if one code page denotes data in an encoding scheme that is different from the other code page, the result is UCS-2 over UTF-8 (that is, the graphic data type over the character data type). (In a non-Unicode database, conversion between different encoding schemes is not supported.)
- For operands that are host variables (whose code page is not BIT DATA), the result code page is the database code page. Input data from such host variables is converted from the application code page to the database code page before being used.

Conversions to the code page of the result are performed, if necessary, for:
- An operand of the concatenation operator
- The selected argument of the COALESCE (also NVL and VALUE) scalar function
- The selected argument of the scalar functions GREATEST, LEAST, MAX, and MIN
- The source-string and insert-string arguments of the scalar function OVERLAY (and INSERT)
Rules for string conversions

- The selected result expression of the CASE expression and the DECODE scalar function
- The expressions of the in list of the IN predicate
- The corresponding expressions of a multiple row VALUES clause
- The corresponding columns involved in set operations.

Character conversion is necessary if all of the following are true:
- The code pages are different
- Neither string is BIT DATA
- The string is neither null nor empty

Examples

Example 1: Given the following in a database created with code page 850:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type</th>
<th>Code Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>COL_1</td>
<td>column</td>
<td>850</td>
</tr>
<tr>
<td>HV_2</td>
<td>host variable</td>
<td>437</td>
</tr>
</tbody>
</table>

When evaluating the predicate:

```
COL_1 CONCAT :HV_2
```

the result code page of the two operands is 850, because the host variable data will be converted to the database code page before being used.

Example 2: Using information from the previous example when evaluating the predicate:

```
COALESCE(COL_1, :HV_2:NULLIND,)
```

the result code page is 850; therefore, the result code page for the COALESCE scalar function will be code page 850.

String comparisons in a Unicode database

Pattern matching is one area where the behavior of existing MBCS databases is slightly different from the behavior of a Unicode database.

For MBCS databases in DB2 Database for Linux, UNIX, and Windows, the current behavior is as follows: If the match-expression contains MBCS data, the pattern can include both SBCS and non-SBCS characters. The special characters in the pattern are interpreted as follows:
- An SBCS halfwidth underscore refers to one SBCS character.
- A non-SBCS fullwidth underscore refers to one non-SBCS character.
- A percent (either SBCS halfwidth or non-SBCS fullwidth) refers to zero or more SBCS or non-SBCS characters.

In a Unicode database, there is really no distinction between "single-byte" and "non-single-byte" characters. Although the UTF-8 format is a "mixed-byte" encoding of Unicode characters, there is no real distinction between SBCS and non-SBCS characters in UTF-8. Every character is a Unicode character, regardless of the number of bytes in UTF-8 format. In a Unicode graphic string, every non-supplementary character, including the halfwidth underscore (U+005F) and
String comparisons in a Unicode database

halfwidth percent (U+0025), is two bytes in width. For Unicode databases, the special characters in the pattern are interpreted as follows:

• For character strings, a halfwidth underscore (X'5F') or a fullwidth underscore (X'EFBCBF') refers to one Unicode character. A halfwidth percent (X'25') or a fullwidth percent (X'EFBC85') refers to zero or more Unicode characters.

• For graphic strings, a halfwidth underscore (U+005F) or a fullwidth underscore (U+FF3F) refers to one Unicode character. A halfwidth percent (U+0025) or a fullwidth percent (U+FF05) refers to zero or more Unicode characters.

Note: Two underscores are needed to match a Unicode supplementary graphic character because such a character is represented by two UCS-2 characters in a graphic string. Only one underscore is needed to match a Unicode supplementary character in a character string.

For the optional “escape expression”, which specifies a character to be used to modify the special meaning of the underscore and percent sign characters, the expression can be specified by any one of:

• A constant
• A special register
• A host variable
• A scalar function whose operands are any of the above
• An expression concatenating any of the above

with the restrictions that:

• No element in the expression can be of type CLOB or DBCLOB. In addition, it cannot be a BLOB file reference variable.

• For character strings, the result of the expression must be one character or a FOR BIT DATA string containing exactly one (1) byte (SQLSTATE 22019). For graphic strings, the result of the expression must be one character (SQLSTATE 22019).
Resolving the anchor object for an anchored type

The anchor object of an anchored type that does not specify ROW is specified with a name that could represent an SQL variable, an SQL parameter, a global variable, a module variable, a column of a table, or a column of a view.

The way the anchor object is resolved depends on the number of identifiers in the anchor object name and the context of the statement using the ANCHOR clause.

- If the anchor object name is specified with 1 identifier, the name could represent an SQL variable, an SQL parameter, a module variable, or a global variable.
- If the anchor object name is specified with 2 identifiers, the name could represent a label-qualified SQL variable, a routine-qualified SQL parameter, a schema-qualified global variable, a module variable, the column of a table, or the column of a view.
- If the anchor object name is specified with 3 identifiers, the name represents a column of a schema-qualified table, a column of a schema-qualified view, a global variable qualified by the current server name and a schema, or a schema-qualified module variable.

An SQL variable is a candidate for an anchor object name only if the ANCHOR clause is used in an SQL variable declaration within a compound statement. An SQL parameter is a candidate for an anchor object name only if the ANCHOR clause is used in an SQL variable declaration within a compound statement used in an SQL routine body.

Resolving the anchor object name that has 1 identifier is done using the following steps:
1. If the ANCHOR clause is in an SQL variable declaration of a compound statement, search for a matching SQL variable name starting from the innermost nested compound to the outermost compound.
2. If the ANCHOR clause is in an SQL variable declaration of a compound statement within a routine body, search for a matching SQL parameter name for the routine.
3. If the ANCHOR clause is used in defining a module object, then search for a matching module variable name within the module.
4. If not yet found, then search for a table or view using the first identifier as the schema name and the second identifier as the table or view name.
5. If not yet found, then search for a global variable with a matching global variable name on the SQL path.

Resolving the anchor object name that has 2 identifiers is done using the following steps:
1. If the ANCHOR clause is in an SQL variable declaration of a compound statement, search for a matching qualified SQL variable name starting from the innermost nested compound to the outermost compound.
2. If the ANCHOR clause is in an SQL variable declaration of a compound statement within a routine body, search for a matching SQL parameter name for the routine if the first identifier of the anchor object name matches the name of the routine.
3. If the ANCHOR clause is used in defining a module object and if the first identifier matches the module name of that module, then search for a module variable name within the module that matches the second identifier.
Resolving the anchor object for an anchored type

4. If not yet found, then search for a table or view column in the current schema using the first identifier as a table or view name and the second identifier as a column name.

5. If not yet found, then search for a global variable using the first identifier as a schema name and the second identifier as a global variable name.

6. If not found and a module was not searched in step 3, then search for a module on the SQL path with a name that matches the first identifier. If found, then use the second identifier to search for a matching published module variable name in the module.

7. If a module is not found using the SQL path in step 6, check for a module public alias that matches the name of the first identifier. If found, then use the second identifier to search for a matching published module variable name in the module identified by the module public alias.

Resolving the anchor object name that has 3 identifiers is done using the following steps:

1. If the ANCHOR clause is used in defining a module object and if the first 2 identifiers match the schema name and the module name of that module, then search for a module variable with a name that matches the last identifier.

2. If not found in the previous step or the step is not applicable, then search for a table or view column using the first identifier as a schema name, the second identifier as a table or view name and the third identifier as a column name.

3. If not found in the previous step and the first identifier is the same as the current server name, then search for a global variable using the second identifier as a schema name, and the third identifier as a global variable name.

4. If not found and a module was not searched in step 1, then search for a published module variable using the first identifier as a schema name, the second identifier as a module name and, if such a module exists, use the third identifier to search for a matching published module variable name in the module.
Resolving the anchor object for an anchored row type

Resolving the anchor object for an anchored row type

The anchor object of an anchored type that includes the ROW keyword is specified with a name that could represent an SQL variable, an SQL parameter, a global variable, a module variable, a table, or a view depending on the context and the number of identifiers in the name and the context of the ANCHOR clause.

The way the anchor object is resolved depends on the number of identifiers in the anchor object name and the context of the statement using the ANCHOR clause.

- If the anchor object name is specified with 1 identifier, the name could represent an SQL variable, an SQL parameter, a module variable, a global variable, a table, or a view.
- If the anchor object name is specified with 2 identifiers, the name could represent a label-qualified SQL variable, a routine-qualified SQL parameter, a schema-qualified global variable, a module variable, a schema-qualified table, or a schema-qualified view.
- If the anchor object name is specified with 3 identifiers, the name could represent a global variable qualified by the current server name and a schema, a table qualified by the current server name and a schema, a view qualified by the current server name and a schema, or a schema-qualified module variable.

An SQL variable is a candidate for an anchor object name only if the ANCHOR clause is used in an SQL variable declaration within a compound statement. An SQL parameter is a candidate for an anchor object name only if the ANCHOR clause is used in an SQL variable declaration within a compound statement used in an SQL routine body. Resolving the anchor object name that has 1 identifier is done using the following steps:

1. If the ANCHOR clause is in an SQL variable declaration of a compound statement, search for a matching SQL variable name starting from the innermost nested compound to the outermost compound.
2. If the ANCHOR clause is in an SQL variable declaration of a compound statement within a routine body, search for a matching SQL parameter name for the routine.
3. If the ANCHOR clause is used in defining a module object, then search for a matching module variable name within the module.
4. If not yet found, then search for a table or view with a matching name in the current schema.
5. If not yet found, then search for a schema global variable with a matching global variable name on the SQL path.

Resolving the anchor object name that has 2 identifiers is done using the following steps:

1. If the ANCHOR clause is in an SQL variable declaration of a compound statement, search for a matching qualified SQL variable name starting from the innermost nested compound to the outermost compound.
2. If the ANCHOR clause is in an SQL variable declaration of a compound statement within a routine body, search for a matching SQL parameter name for the routine if the first identifier of the anchor object name matches the name of the routine.
3. If the ANCHOR clause is used in defining a module object and if the first identifier matches the module name of that module, then search for a module variable name within the module that matches the second identifier.
Resolving the anchor object for an anchored row type

4. If not yet found, then search for a table or view using the first identifier as the schema name and the second identifier as the table or view name.

5. If not yet found, then search for a global variable using the first identifier as a schema name and the second identifier as a global variable name.

6. If not found and a module was not searched in step 3, then search for a module on the SQL path with a name that matches the first identifier. If found, then use the second identifier to search for a matching published module variable name in the module.

7. If a module is not found using the SQL path in step 6, check for a module public alias that matches the name of the first identifier. If found, then use the second identifier to search for a matching published module variable name in the module identified by the module public alias.

Resolving the anchor object name that has 3 identifiers is done using the following steps:

1. If the ANCHOR clause is used in defining a module object and if the first 2 identifiers match the schema name and the module name of that module, then search for a module variable with a name that matches the last identifier.

2. If not found and the first identifier is the same as the current server name, then search for a table or view using the second identifier as the schema name and the third identifier as the table or view name.

3. If not found and the first identifier is the same as the current server name, then search for a global variable using the second identifier as the schema name and the third identifier as the global variable name.

4. If not found and a module was not searched in step 1, then search for a published module variable using the first identifier as a schema name, the second identifier as a module name and, if such a module exists, use the third identifier to search for a matching published variable name in the module.
Database partition-compatible data types

Database partition compatibility is defined between the base data types of corresponding columns of distribution keys. Database partition-compatible data types have the property that two variables, one of each type, with the same value, are mapped to the same distribution map index by the same database partitioning function.

Table 17 shows the compatibility of data types in database partitions.

Database partition compatibility has the following characteristics:

- Internal formats are used for DATE, TIME, and TIMESTAMP. They are not compatible with each other, and none are compatible with character or graphic data types.
- Partition compatibility is not affected by the nullability of a column.
- Partition compatibility is affected by collation. Locale-sensitive UCA-based collations require an exact match in collation, except that the strength (S) attribute of the collation is ignored. All other collations are considered equivalent for the purposes of determining partition compatibility.
- Character columns defined with FOR BIT DATA are only compatible with character columns without FOR BIT DATA when a collation other than a locale-sensitive UCA-based collation is used.
- NULL values of compatible data types are treated identically. Different results might be produced for NULL values of non-compatible data types.
- Base data type of the UDT is used to analyze database partition compatibility.
- Timestamps of the same value in the distribution key are treated identically, even if their timestamp precisions differ.
- Decimals of the same value in the distribution key are treated identically, even if their scale and precision differ.
- Trailing blanks in character strings (CHAR, VARCHAR, GRAPHIC or VARGRAPHIC) are ignored by the system-provided hashing function.
- When a locale-sensitive UCA-based collation is used, CHAR, VARCHAR, GRAPHIC, and VARGRAPHIC are compatible data types. When other collations are used, CHAR and VARCHAR are compatible types and GRAPHIC and VARGRAPHIC are compatible types, but CHAR and VARCHAR are not compatible types with GRAPHIC and VARGRAPHIC. CHAR or VARCHAR of different lengths are compatible data types.
- DECFLOAT values that are equal are treated identically even if their precision differs. DECFLOAT values that are numerically equal are treated identically even if they have a different number of significant digits.
- Data types that are not supported as part of a distribution key are not applicable for database partition compatibility. This includes columns whose data type is BLOB, CLOB, DBCLOB, XML, distinct type based on any of these data types, or structured type.

Table 17. Database Partition Compatibilities

| Operands     | Binary | Integer | Decimal | Number | Floating-point | Decimal Floating-point | Character String | Graphic String | Date | Time | Time- | Distinct Type |
|--------------|--------|---------|---------|--------|----------------|-----------------------|-----------------|----------------|------|------|stamp |              |
| Binary       | Yes    | No      | No      | No     | No             | No                    | No              | No             | No   | No   | No    | 1              |
| Integer      |        |         |         |        |                |                       |                 |                |      |      |       |               |
| Decimal      | No     | Yes     | No      | No     | No             | No                    | No              | No             | No   | No   | No    | 1              |
| Number       |        |         |         |        |                |                       |                 |                |      |      |       |               |
### Table 17. Database Partition Compatibilities (continued)

<table>
<thead>
<tr>
<th>Operands</th>
<th>Binary</th>
<th>Decimal</th>
<th>Floating-point</th>
<th>Decimal Floating-point</th>
<th>Character String</th>
<th>Graphic String</th>
<th>Date</th>
<th>Time</th>
<th>Time-stamp</th>
<th>Distinct Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Floating-point</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes 1</td>
</tr>
<tr>
<td>Decimal Floating-point</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes 1</td>
</tr>
<tr>
<td>Character String</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes 2</td>
<td>2, 3</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes 1</td>
</tr>
<tr>
<td>Graphic String</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes 2</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes 1</td>
</tr>
<tr>
<td>Date</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes 1</td>
</tr>
<tr>
<td>Time</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes 1</td>
</tr>
<tr>
<td>Timestamp</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Distinct Type</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Note:**

1. A distinct type value is database partition compatible with the source data type of the distinct type or with any other distinct type with the same source data type. The source data type of the distinct type must be a data type that is supported as part of a distribution key. A user-defined distinct type (UDT) value is database partition compatible with the source type of the UDT or any other UDT with a database partition compatible source type. A distinct type cannot be based on BLOB, CLOB, DBCLOB, or XML.

2. Character and graphic string types are compatible when they have compatible collations.

3. Character and graphic string types are compatible when a locale-sensitive UCA-based collation is in effect. Otherwise, they are not compatible types.
**Constants**

A *constant* (sometimes called a *literal*) specifies a value. Constants are classified as string constants or numeric constants. Numeric constants are further classified as integer, floating-point, or decimal.

All constants have the NOT NULL attribute.

A negative zero value in a numeric constant (-0) is the same value as a zero without the sign (0).

User-defined types have strong typing. This means that a user-defined type is only compatible with its own type. A constant, however, has a built-in type. Therefore, an operation involving a user-defined type and a constant is only possible if the user-defined type has been cast to the constant’s built-in type, or if the constant has been cast to the user-defined type. For example, using the table and distinct type in “User-defined type comparisons” on page 128, the following comparisons with the constant 14 are valid:

```
SELECT * FROM CAMP_DB2_ROSTER
WHERE AGE > CAST(14 AS YOUTH)
SELECT * FROM CAMP_DB2_ROSTER
WHERE CAST(AGE AS INTEGER) > 14
```

The following comparison is not valid:

```
SELECT * FROM CAMP_DB2_ROSTER
WHERE AGE > 14
```

**Integer constants**

An *integer constant* specifies an integer as a signed or unsigned number with a maximum of 19 digits that does not include a decimal point. The data type of an integer constant is large integer if its value is within the range of a large integer. The data type of an integer constant is big integer if its value is outside the range of large integer but within the range of a big integer. A constant that is defined outside the range of big integer values is considered a decimal constant.

Note that the smallest literal representation of a large integer constant is -2 147 483 647, and not -2 147 483 648, which is the limit for integer values. Similarly, the smallest literal representation of a big integer constant is -9 223 372 036 854 775 807, and not -9 223 372 036 854 775 808, which is the limit for big integer values.

*Examples:*

64    -15    +100    32767    720176    12345678901

In syntax diagrams, the term ‘integer’ is used for a large integer constant that must not include a sign.

**Floating-point constants**

A *floating-point constant* specifies a floating-point number as two numbers separated by an E. The first number may include a sign and a decimal point; the second number may include a sign but not a decimal point. The data type of a floating-point constant is double-precision. The value of the constant is the product
of the first number and the power of 10 specified by the second number; it must be within the range of floating-point numbers. The number of bytes in the constant must not exceed 30.

Examples:

15E1  2.E5  2.2E-1   +5.E+2

Decimal constants

A decimal constant is a signed or unsigned number that consists of no more than 31 digits and either includes a decimal point or is not within the range of binary integers. It must be within the range of decimal numbers. The precision is the total number of digits (including leading and trailing zeros); the scale is the number of digits to the right of the decimal point (including trailing zeros).

Examples:

25.5  1000.  -15.  +37589.333333333

Decimal floating-point constants

There are no decimal floating-point constants except for the decimal floating-point special values, which are interpreted as DECFLOAT(34).

These special values are: INFINITY, NAN, and SNAN. INFINITY represents infinity, a number whose magnitude is infinitely large. INFINITY can be preceded by an optional sign. INF can be specified in place of INFINITY. NAN represents Not a Number (NaN) and is sometimes called quiet NaN. It is a value that represents undefined results which does not cause a warning or exception. SNAN represents signaling NaN (sNaN). It is a value that represents undefined results which will cause a warning or exception if used in any operation that is defined in any numerical operation. Both NAN and SNAN can be preceded by an optional sign, but the sign is not significant for arithmetic operations. SNAN can be used in non-numerical operations without causing a warning or exception, for example in the VALUES list of an INSERT or as a constant compared in a predicate.

SNaN -INFINITY

When one of the special values (INFINITY, NAN, and SNAN) is used in a context where it could be interpreted as a name, explicitly cast the value to decimal-floatin point. For example:

CAST ('SNaN' AS DECFLOAT)

All non-special values are interpreted as integer, floating-point or decimal constants, in accordance with the rules specified above. To obtain a numeric decimal floating-point value, use the DECFLOAT cast function with a character string constant. It is not recommended to use floating-point constants as arguments to the DECFLOAT function, because floating-point is not exact and the resulting decimal floating-point value might be different than the decimal digit characters that make up the string. Instead, use character constants as arguments to the DECFLOAT function.

For example, DECFLOAT('6.0221415E23', 34) returns the decimal floating-point value 6.0221415E+23, but DECFLOAT(6.0221415E23, 34) returns the decimal floating-point value 6.0221415000000003E+23.
Character string constants

A **character string constant** specifies a varying-length character string. There are three forms of a character string constant:

- A sequence of characters that starts and ends with a string delimiter, which is an apostrophe ('). The number of bytes between the string delimiters cannot be greater than 32,672. Two consecutive string delimiters are used to represent one string delimiter within the character string. Two consecutive string delimiters that are not contained within a string represent the empty string.

- X followed by a sequence of characters that starts and ends with a string delimiter. This form of a character string constant is also called a *hexadecimal constant*. The characters between the string delimiters must be an even number of hexadecimal digits. Blanks between the string delimiters are ignored. The number of hexadecimal digits must not exceed 32,672. A hexadecimal digit is a digit or any of the letters A through F (uppercase or lowercase). Under the conventions of hexadecimal notation, each pair of hexadecimal digits represents a character. This form of a character string constant allows you to specify characters that do not have a keyboard representation.

- U& followed by a sequence of characters that starts and ends with a string delimiter and that is optionally followed by the UESCAPE clause. This form of a character string constant is also called a *Unicode string constant*. The number of bytes between the string delimiters cannot be greater than 32,672. The Unicode string constant is converted from UTF-8 to the section code page during statement compilation. Two consecutive string delimiters are used to represent one string delimiter within the character string. Two consecutive Unicode escape characters are used to represent one Unicode escape character within the character string, but these characters count as one character when calculating the lengths of character constants. Two consecutive string delimiters that are not contained within a string represent the empty string. Because a character in UTF-8 can range from 1 to 4 bytes, a Unicode string constant of the maximum length might actually represent fewer than 32,672 characters.

A character can be expressed by either its typographical character (glyph) or its Unicode code point. The code point of a Unicode character ranges from X’000000’ to X’10FFFF’. To express a Unicode character through its code point, use the Unicode escape character followed by 4 hexadecimal digits, or the Unicode escape character followed by a plus sign (+) and 6 hexadecimal digits. The default Unicode escape character is the reverse solidus (\), but a different character can be specified with the UESCAPE clause. The UESCAPE clause is specified as the UESCAPE keyword followed by a single character between string delimiters. The Unicode escape character cannot be a plus sign (+), a double quotation mark ("), a single quotation mark ('), a blank, or any of the characters 0 through 9 or A through F, in either uppercase or lowercase (SQLSTATE 42604). An example of the two ways in which the Latin capital letter A can be specified as a Unicode code point is \0041 and \+000041.

The constant value is always converted to the database code page when it is bound to the database. It is considered to be in the database code page. Therefore, if used in an expression that combines a constant with a FOR BIT DATA column, and whose result is FOR BIT DATA, the constant value will not be converted from its database code page representation when used.

**Examples:**

```
'12/14/1985'  '32'  'DON''T CHANGE'  ''
X'FFFF'   X'46 72 61 6E 6B'
U&'01416d\017A is a city in Poland'  U&'c:\temp'  U&'01D11E'  UESCAPE '0'
```
The rightmost string on the second line in the example represents the VARCHAR pattern of the ASCII string ‘Frank’. The last line corresponds to: ‘óóódó’ is a city in Poland’, ‘c:\temp’, and a single character representing the musical symbol G clef.

**Graphic string constants**

A graphic string constant specifies a varying-length graphic string consisting of a sequence of double-byte characters that starts and ends with a single-byte apostrophe (‘), and that is preceded by a single-byte G or N. The characters between the apostrophes must represent an even number of bytes, and the length of the graphic string must not exceed 16,336 bytes.

*Examples:*

```plaintext
G'double-byte character string'
N'double-byte character string'
```

The apostrophe must not appear as part of an MBCS character to be considered a delimiter.

In a Unicode database, a hexadecimal graphic string constant that specifies a varying-length graphic string is also supported. The format of a hexadecimal graphic string constant is: GX followed by a sequence of characters that starts and ends with an apostrophe. The characters between the apostrophes must be an even multiple of four hexadecimal digits. The number of hexadecimal digits must not exceed 16,336; otherwise, an error is returned (SQLSTATE 54002). If a hexadecimal graphic string constant is improperly formed, an error is returned (SQLSTATE 42606). Each group of four digits represents a single graphic character. In a Unicode database, this would be a single UCS-2 graphic character.

*Examples:*

```plaintext
GX'FFFF'
```

represents the bit pattern ‘1111111111111111’ in a Unicode database.

```plaintext
GX'005200690063006B'
```

represents the VARGRAPHIC pattern of the ASCII string ‘Rick’ in a Unicode database.

**Datetime constants**

A datetime constant specifies a date, time, or timestamp.

Typically, character-string constants are used to represent constant datetime values in assignments and comparisons. However, the associated data type name can be used preceding specific formats of the character-string constant to specifically denote the constant as a datetime constant instead of a character-string constant. The format for the three datetime constants are:

```plaintext
DATE 'yyyy-mm-dd'
```

The data type of the value is DATE.

```plaintext
TIME 'hh:mm:ss'
```

or

```plaintext
TIME 'hh:mm'
```

The data type of the value is TIME.

```plaintext
TIMESTAMP 'yyyy-mm-dd hh:mm:ss.nnnnnnnnnnnn'
```

Chapter 2. Language elements 147
or

**TIMESTAMP 'yyyy-mm-dd-hh.mm.ss.nnnnnnnnnnnn'**

where the number of digits of fractional seconds can vary from 0 to 12 and the period character can be omitted if there are no fractional seconds. The data type of the value is `TIMESTAMP(p)`, where `p` is the number of digits of fractional seconds.

Leading zeros can be omitted from the month, day, and hour part of the character-string constant portion, where applicable, in each of these datetime constants. Leading zero characters must be included for minutes and seconds elements of `TIME` or `TIMESTAMP` constants. Trailing blanks can be included and are ignored.

**UCS-2 graphic string constants**

In a Unicode database, a hexadecimal UCS-2 graphic string that specifies a varying-length UCS-2 graphic string constant is supported. The format of a hexadecimal UCS-2 graphic string constant is: UX followed by a sequence of characters that starts and ends with an apostrophe. The characters between the apostrophes must be an even multiple of four hexadecimal digits. The number of hexadecimal digits must not exceed 16336; otherwise, an error is returned (SQLSTATE 54002). If a hexadecimal UCS-2 graphic string constant is improperly formed, an error is returned (SQLSTATE 42606). Each group of four digits represents a single UCS-2 graphic character.

*Example:*

```sql
UX '0042006F006200620079'
```

represents the VARGRAPHIC pattern of the ASCII string 'Bobby'.

**Boolean constants**

A Boolean constant specifies the keyword TRUE or FALSE, representing the truth values true and false respectively. The unknown truth value can be specified using `CAST(NULL AS BOOLEAN)`. 

---

**Constants**

...
A special register is a storage area that is defined for an application process by the database manager. It is used to store information that can be referenced in SQL statements. A reference to a special register is a reference to a value provided by the current server. If the value is a string, its CCSID is a default CCSID of the current server. The special registers can be referenced as follows:
Notes:

1. The SQL2003 Core standard uses the form with the underscore.
Some special registers can be updated using the SET statement. The following table shows which of the special registers can be updated as well as indicating which special register can be the null value.

**Table 18. Updatable and nullable special registers**

<table>
<thead>
<tr>
<th>Special Register</th>
<th>Updatable</th>
<th>Nullable</th>
</tr>
</thead>
<tbody>
<tr>
<td>CURRENT_CLIENT_ACCTNG</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>CURRENT_CLIENT_APPLNAME</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>CURRENT_CLIENT_USERID</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>CURRENT_CLIENT_WRKSTNNAME</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>CURRENT_DATE</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>CURRENT_DBPARTITIONNUM</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>CURRENT_DECFLOAT_ROUNDING_MODE</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>CURRENT_DEFAULT_TRANSFORM_GROUP</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>CURRENT_DEGREE</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>CURRENT_EXPLAIN_MODE</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>CURRENT_EXPLAIN_SNAPSHOT</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>CURRENT_FEDERATEDASYNCHRONY</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>CURRENT_IMPLICIT_XMLPARSE_OPTION</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>CURRENT_ISOATION</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>“CURRENT_LOCALE LC_MESSAGES” on page 167</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>CURRENT_LOCALE LC_TIME</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>CURRENT_LOCK_TIMEOUT</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>CURRENT_MAINTAINED_TABLE_TYPES_FOR_OPTIMIZATION</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>CURRENT_MDC_ROLLOUT_MODE</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>CURRENT_OPTIMIZATION_PROFILE</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>CURRENT_PACKAGE_PATH</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>CURRENT_PATH</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>CURRENT_QUERY_OPTIMIZATION</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>CURRENT_REFRESH_AGE</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>CURRENT_SCHEMA</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>CURRENT_SERVER</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>CURRENT_SQL_CCFLAGS</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>CURRENT_TIME</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>CURRENT_TIMESTAMP</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>CURRENT_TIMEZONE</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>CURRENT_USER</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>SESSION_USER</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>SYSTEM_USER</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>USER</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

When a special register is referenced in a routine, the value of the special register in the routine depends on whether the special register is updatable or not. For non-updatable special registers, the value is set to the default value for the special...
Special registers

register. For updatable special registers, the initial value is inherited from the invoker of the routine and can be changed with a subsequent SET statement inside the routine.
CURRENT CLIENT_ACCTNG

CURRENT CLIENT_ACCTNG

The CURRENT CLIENT_ACCTNG (or CLIENT ACCTNG) special register contains the value of the accounting string from the client information specified for this connection. The data type of the register is VARCHAR(255). The default value of this register is an empty string.

The value of the accounting string can be changed by using the Set Client Information (sqleseti) API.

Note that the value provided via the sqleseti API is in the application code page, and the special register value is stored in the database code page. Depending on the data values used when setting the client information, truncation of the data value stored in the special register may occur during code page conversion.

Example: Get the current value of the accounting string for this connection.

```sql
VALUES (CURRENT CLIENT_ACCTNG)
INTO :ACCT_STRING
```
The CURRENT CLIENT_APPLNAME (or CLIENT APPLNAME) special register contains the value of the application name from the client information specified for this connection. The data type of the register is VARCHAR(255). The default value of this register is an empty string.

The value of the application name can be changed by using the Set Client Information (sqleseti) API.

Note that the value provided via the sqleseti API is in the application code page, and the special register value is stored in the database code page. Depending on the data values used when setting the client information, truncation of the data value stored in the special register may occur during code page conversion.

Example: Select which departments are allowed to use the application being used in this connection.

```sql
SELECT DEPT
FROM DEPT_APPL_MAP
WHERE APPL_NAME = CURRENT CLIENT_APPLNAME
```
CURRENT CLIENT_USERID

The CURRENT CLIENT_USERID (or CLIENT_USERID) special register contains the value of the client user ID from the client information specified for this connection. The data type of the register is VARCHAR(255). The default value of this register is an empty string.

The value of the client user ID can be changed by using the Set Client Information (sqleseti) API.

Note that the value provided via the sqleseti API is in the application code page, and the special register value is stored in the database code page. Depending on the data values used when setting the client information, truncation of the data value stored in the special register may occur during code page conversion.

Example: Find out in which department the current client user ID works.

```sql
SELECT DEPT
FROM DEPT_USERID_MAP
WHERE USER_ID = CURRENT_CLIENT_USERID
```
CURRENT CLIENT_WRKSTNNNAME

The CURRENT CLIENT_WRKSTNNNAME (or CLIENT WRKSTNNNAME) special register contains the value of the workstation name from the client information specified for this connection. The data type of the register is VARCHAR(255). The default value of this register is an empty string.

The value of the workstation name can be changed by using the Set Client Information (sqleseti) API.

Note that the value provided via the sqleseti API is in the application code page, and the special register value is stored in the database code page. Depending on the data values used when setting the client information, truncation of the data value stored in the special register may occur during code page conversion.

Example: Get the workstation name being used for this connection.

```
VALUES (CURRENT CLIENT_WRKSTNNNAME)
INTO :WS_NAME
```
The CURRENT DATE (or CURRENT_DATE) special register specifies a date that is based on a reading of the time-of-day clock when the SQL statement is executed at the application server. If this special register is used more than once within a single SQL statement, or used with CURRENT TIME or CURRENT TIMESTAMP within a single statement, all values are based on a single clock reading.

When used in an SQL statement inside a routine, CURRENT DATE is not inherited from the invoking statement.

In a federated system, CURRENT DATE can be used in a query intended for data sources. When the query is processed, the date returned will be obtained from the CURRENT DATE register at the federated server, not from the data sources.

Example: Run the following command from the DB2 CLP to obtain the current date.

```
db2 values CURRENT DATE
```

Example: Using the PROJECT table, set the project end date (PRENDATE) of the MA2111 project (PROJNO) to the current date.

```
UPDATE PROJECT
    SET PRENDATE = CURRENT DATE
WHERE PROJNO = 'MA2111'
```
CURRENT DBPARTITIONNUM

The CURRENT DBPARTITIONNUM special register specifies an INTEGER value that identifies the coordinator node number for the statement. For statements issued from an application, the coordinator is the database partition to which the application connects. For statements issued from a routine, the coordinator is the database partition from which the routine is invoked.

When used in an SQL statement inside a routine, CURRENT DBPARTITIONNUM is never inherited from the invoking statement.

CURRENT DBPARTITIONNUM returns 0 if the database instance is not defined to support database partitioning. (In other words, if there is no db2nodes.cfg file. For partitioned databases, the db2nodes.cfg file exists and contains database partition definitions.)

CURRENT DBPARTITIONNUM can be changed through the CONNECT statement, but only under certain conditions.

For compatibility with versions earlier than Version 8, the keyword NODE can be substituted for DBPARTITIONNUM.

Example: Set the host variable APPL_NODE (integer) to the number of the database partition to which the application is connected.

```sql
VALUES CURRENT DBPARTITIONNUM
    INTO :APPL_NODE
```
The CURRENT DECFLOAT ROUNDING MODE special register specifies the rounding mode that is used for DECFLOAT values.

The data type is VARCHAR(128). The following rounding modes are supported:

- ROUND_CEILING rounds the value towards positive infinity. If all of the discarded digits are zero or if the sign is negative, the result is unchanged (except for the removal of the discarded digits). Otherwise, the result coefficient is incremented by 1.
- ROUND_DOWN rounds the value towards 0 (truncation). The discarded digits are ignored.
- ROUND_FLOOR rounds the value towards negative infinity. If all of the discarded digits are zero or if the sign is positive, the result is unchanged (except for the removal of the discarded digits). Otherwise, the sign is negative and the result coefficient is incremented by 1.
- ROUND_HALF_EVEN rounds the value to the nearest value. If the values are equidistant, rounds the value so that the final digit is even. If the discarded digits represent more than half of the value of a number in the next left position, the result coefficient is incremented by 1. If they represent less than half, the result coefficient is not adjusted (that is, the discarded digits are ignored). Otherwise, the result coefficient is unaltered if its rightmost digit is even, or incremented by 1 if its rightmost digit is odd (to make an even digit).
- ROUND_HALF_UP rounds the value to the nearest value. If the values are equidistant, rounds the value up. If the discarded digits represent half or more than half of the value of a number in the next left position, the result coefficient is incremented by 1. Otherwise, the discarded digits are ignored.

The value of the DECFLOAT rounding mode on a client can be confirmed to match that of the server by invoking the SET CURRENT DECFLOAT ROUNDING MODE statement. However, this statement cannot be used to change the rounding mode of the server. The initial value of CURRENT DECFLOAT ROUNDING MODE is determined by the decflt_rounding database configuration parameter and can only be changed by changing the value of this database configuration parameter.
CURRENT DEFAULT TRANSFORM GROUP

The CURRENT DEFAULT TRANSFORM GROUP special register specifies a VARCHAR(18) value that identifies the name of the transform group used by dynamic SQL statements for exchanging user-defined structured type values with host programs. This special register does not specify the transform groups used in static SQL statements, or in the exchange of parameters and results with external functions or methods.

Its value can be set by the SET CURRENT DEFAULT TRANSFORM GROUP statement. If no value is set, the initial value of the special register is the empty string (a VARCHAR with a length of zero).

In a dynamic SQL statement (that is, one which interacts with host variables), the name of the transform group used for exchanging values is the same as the value of this special register, unless this register contains the empty string. If the register contains the empty string (no value was set by using the SET CURRENT DEFAULT TRANSFORM GROUP statement), the DB2_PROGRAM transform group is used for the transform. If the DB2_PROGRAM transform group is not defined for the structured type subject, an error is raised at run time (SQLSTATE 42741).

Examples:

Set the default transform group to MYSTRUCT1. The TO SQL and FROM SQL functions defined in the MYSTRUCT1 transform are used to exchange user-defined structured type variables with the host program.

```
SET CURRENT DEFAULT TRANSFORM GROUP = MYSTRUCT1
```

Retrieve the name of the default transform group assigned to this special register.

```
VALUES (CURRENT DEFAULT TRANSFORM GROUP)
```
CURRENT DEGREE

The CURRENT DEGREE special register specifies the degree of intra-partition parallelism for the execution of dynamic SQL statements. (For static SQL, the DEGREE bind option provides the same control.) The data type of the register is CHAR(5). Valid values are ANY or the string representation of an integer between 1 and 32 767, inclusive.

If the value of CURRENT DEGREE represented as an integer is 1 when an SQL statement is dynamically prepared, the execution of that statement will not use intra-partition parallelism.

If the value of CURRENT DEGREE represented as an integer is greater than 1 and less than or equal to 32 767 when an SQL statement is dynamically prepared, the execution of that statement can involve intra-partition parallelism with the specified degree.

If the value of CURRENT DEGREE is ANY when an SQL statement is dynamically prepared, the execution of that statement can involve intra-partition parallelism using a degree determined by the database manager.

The actual runtime degree of parallelism will be the lower of:

- The value of the maximum query degree (max_querydegree) configuration parameter
- The application runtime degree
- The SQL statement compilation degree.

If the intra_parallel database manager configuration parameter is set to NO, the value of the CURRENT DEGREE special register will be ignored for the purpose of optimization, and the statement will not use intra-partition parallelism.

The value can be changed by invoking the SET CURRENT DEGREE statement.

The initial value of CURRENT DEGREE is determined by the dft_degree database configuration parameter.
CURRENT EXPLAIN MODE

The CURRENT EXPLAIN MODE special register holds a VARCHAR(254) value which controls the behavior of the Explain facility with respect to eligible dynamic SQL statements. This facility generates and inserts Explain information into the Explain tables. This information does not include the Explain snapshot. Possible values are YES, EXPLAIN, NO, REOPT, RECOMMEND INDEXES, and EVALUATE INDEXES. (For static SQL, the EXPLAIN bind option provides the same control. In the case of the PREP and BIND commands, the EXPLAIN option values are: YES, NO, and ALL.)

YES Enables the Explain facility and causes Explain information for a dynamic SQL statement to be captured when the statement is compiled.

EXPLAIN Enables the facility, but dynamic statements are not executed.

NO Disables the Explain facility.

REOPT Enables the Explain facility and causes Explain information for a dynamic (or incremental-bind) SQL statement to be captured only when the statement is reoptimized using real values for the input variables (host variables, special registers, global variables, or parameter markers).

RECOMMEND INDEXES Recommends a set of indexes for each dynamic query. Populates the ADVISE_INDEX table with the set of indexes.

EVALUATE INDEXES Explains dynamic queries as though the recommended indexes existed. The indexes are picked up from the ADVISE_INDEX table.

The initial value is NO. The value can be changed by invoking the SET CURRENT EXPLAIN MODE statement.

The CURRENT EXPLAIN MODE and CURRENT EXPLAIN SNAPSHOT special register values interact when the Explain facility is invoked. The CURRENT EXPLAIN MODE special register also interacts with the EXPLAIN bind option. RECOMMEND INDEXES and EVALUATE INDEXES can only be set for the CURRENT EXPLAIN MODE register, and must be set using the SET CURRENT EXPLAIN MODE statement.

Example: Set the host variable EXPL_MODE (VARCHAR(254)) to the value currently in the CURRENT EXPLAIN MODE special register.

VALUES CURRENT EXPLAIN MODE INTO :EXPL_MODE
CURRENT EXPLAIN SNAPSHOT

The CURRENT EXPLAIN SNAPSHOT special register holds a CHAR(8) value that controls the behavior of the Explain snapshot facility. This facility generates compressed information, including access plan information, operator costs, and bind-time statistics.

Only the following statements consider the value of this register: CALL, Compound SQL (Dynamic), DELETE, INSERT, MERGE, REFRESH, SELECT, SELECT INTO, SET INTEGRITY, UPDATE, VALUES, or VALUES INTO. Possible values are YES, EXPLAIN, NO, and REOPT. (For static SQL, the EXPLSNAP bind option provides the same control. In the case of the PREP and BIND commands, the EXPLSNAP option values are: YES, NO, and ALL.)

YES Enables the Explain snapshot facility and takes a snapshot of the internal representation of a dynamic SQL statement as the statement is compiled.

EXPLAIN Enables the Explain snapshot facility, but dynamic statements are not executed.

NO Disables the Explain snapshot facility.

REOPT Enables the Explain facility and causes Explain information for a dynamic (or incremental-bind) SQL statement to be captured only when the statement is reoptimized using real values for the input variables (host variables, special registers, global variables, or parameter markers).

The initial value is NO. The value can be changed by invoking the SET CURRENT EXPLAIN SNAPSHOT statement.

The CURRENT EXPLAIN SNAPSHOT and CURRENT EXPLAIN MODE special register values interact when the Explain facility is invoked. The CURRENT EXPLAIN SNAPSHOT special register also interacts with the EXPLSNAP bind option.

Example: Set the host variable EXPL_SNAP (char(8)) to the value currently in the CURRENT EXPLAIN SNAPSHOT special register.

VALUES CURRENT EXPLAIN SNAPSHOT INTO :EXPL_SNAP
CURRENT FEDERATED ASYNCHRONY

The CURRENT FEDERATED ASYNCHRONY special register specifies the degree of asynchrony for the execution of dynamic SQL statements. (The FEDERATED_ASYNCHRONY bind option provides the same control for static SQL.) The data type of the register is INTEGER. Valid values are ANY (representing -1) or an integer between 0 and 32 767, inclusive. If, when an SQL statement is dynamically prepared, the value of CURRENT FEDERATED ASYNCHRONY is:

- 0, the execution of that statement will not use asynchrony
- Greater than 0 and less than or equal to 32 767, the execution of that statement can involve asynchrony using the specified degree
- ANY (representing -1), the execution of that statement can involve asynchrony using a degree that is determined by the database manager

The value of the CURRENT FEDERATED ASYNCHRONY special register can be changed by invoking the SET CURRENT FEDERATED ASYNCHRONY statement.

The initial value of the CURRENT FEDERATED ASYNCHRONY special register is determined by the federated_async database manager configuration parameter if the dynamic statement is issued through the command line processor (CLP). The initial value is determined by the FEDERATED_ASYNCHRONY bind option if the dynamic statement is part of an application that is being bound.

Example: Set the host variable FEDASYNC (INTEGER) to the value of the CURRENT FEDERATED ASYNCHRONY special register.

VALUES CURRENT FEDERATED ASYNCHRONY INTO :FEDASYNC
CURRENT IMPLICIT XMLPARSE OPTION

The CURRENT IMPLICIT XMLPARSE OPTION special register specifies the whitespace handling options that are to be used when serialized XML data is implicitly parsed by the DB2 server, without validation. An implicit non-validating parse operation occurs when an SQL statement is processing an XML host variable or an implicitly or explicitly typed XML parameter marker that is not an argument of the XMLVALIDATE function. The data type of the register is VARCHAR(19).

The value of the CURRENT IMPLICIT XMLPARSE OPTION special register can be changed by invoking the SET CURRENT IMPLICIT XMLPARSE OPTION statement. Its initial value is 'STRIP WHITESPACE'.

Examples:

Retrieve the value of the CURRENT IMPLICIT XMLPARSE OPTION special register into a host variable named CURXMLPARSEOPT:

```sql
EXEC SQL
VALUES (CURRENT IMPLICIT XMLPARSE OPTION) INTO :CURXMLPARSEOPT;
```

Set the CURRENT IMPLICIT XMLPARSE OPTION special register to 'PRESERVE WHITESPACE'.

```sql
SET CURRENT IMPLICIT XMLPARSE OPTION = 'PRESERVE WHITESPACE'
```

Whitespace is then preserved when the following SQL statement executes:

```sql
INSERT INTO T1 (XMLCOL1) VALUES (?)
```
CURRENT ISOLATION

The CURRENT ISOLATION special register holds a CHAR(2) value that identifies the isolation level (in relation to other concurrent sessions) for any dynamic SQL statements issued within the current session.

The possible values are:

(blanks)  Not set; use the isolation attribute of the package.
UR       Uncommitted Read
CS       Cursor Stability
RR       Repeatable Read
RS       Read Stability

The value of the CURRENT ISOLATION special register can be changed by the SET CURRENT ISOLATION statement.

Until a SET CURRENT ISOLATION statement is issued within a session, or after RESET has been specified for SET CURRENT ISOLATION, the CURRENT ISOLATION special register is set to blanks and is not applied to dynamic SQL statements; the isolation level used is taken from the isolation attribute of the package which issued the dynamic SQL statement. Once a SET CURRENT ISOLATION statement has been issued, the CURRENT ISOLATION special register provides the isolation level for any subsequent dynamic SQL statement compiled within the session, regardless of the settings for the package issuing the statement. This will remain in effect until the session ends or until a SET CURRENT ISOLATION statement is issued with the RESET option.

Example: Set the host variable ISOLATION_MODE (CHAR(2)) to the value currently stored in the CURRENT ISOLATION special register.

VALUES CURRENT ISOLATION
INTO :ISOLATION_MODE
The CURRENT LOCALE LC_MESSAGES special register identifies the locale that is used by monitoring routines in the monreport module.

The monitoring routines use the value of CURRENT LOCALE LC_MESSAGES to determine what language the result set text output from the routines should be returned in. User-defined routines that are coded to return messages could also use the value of CURRENT LOCALE LC_MESSAGES to determine what language to use for message text.

The data type is VARCHAR(128).

The initial value of CURRENT LOCALE LC_MESSAGES is "en_US" for English (United States). The value can be changed by invoking the SET CURRENT LOCALE LC_MESSAGES statement.
CURRENT LOCALE LC_TIME

The CURRENT LOCALE LC_TIME special register identifies the locale that is used for SQL statements that involve the datetime related built-in functions DAYNAME, MONTHNAME, NEXT_DAY, ROUND, ROUND_TIMESTAMP, TIMESTAMP_FORMAT, TRUNCATE, TRUNC_TIMESTAMP and VARCHAR_FORMAT. These functions use the value of CURRENT LOCALE LC_TIME if the locale-name argument is not explicitly specified.

The data type is VARCHAR(128).

The initial value of CURRENT LOCALE LC_TIME is “en_US” for English (United States). The value can be changed by invoking the SET CURRENT LOCALE LC_TIME statement.
CURRENT LOCK TIMEOUT

The CURRENT LOCK TIMEOUT special register specifies the number of seconds to wait for a lock before returning an error indicating that a lock cannot be obtained. This special register impacts row, table, index key, MDC block, and XML path (XPath) locks. The data type of the register is INTEGER.

Valid values for the CURRENT LOCK TIMEOUT special register are integers between -1 and 32767, inclusive. This special register can also be set to the null value. A value of -1 specifies that timeouts are not to take place, and that the application is to wait until the lock is released or a deadlock is detected. A value of 0 specifies that the application is not to wait for a lock; if a lock cannot be obtained, an error is to be returned immediately.

The value of the CURRENT LOCK TIMEOUT special register can be changed by invoking the SET CURRENT LOCK TIMEOUT statement. Its initial value is null; in this case, the current value of the locktimeout database configuration parameter is used when waiting for a lock, and this value will be returned for the special register.
CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION

The CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION special register specifies a VARCHAR(254) value that identifies the types of tables that can be considered when optimizing the processing of dynamic SQL queries. Materialized query tables are never considered by static embedded SQL queries.

The initial value of CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION is SYSTEM. Its value can be changed by the SET CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION statement.
CURRENT MDC ROLLOUT MODE

The CURRENT MDC ROLLOUT MODE special register specifies the behavior on multidimensional clustering (MDC) tables of DELETE statements that qualify for rollout processing.

The default value of this register is determined by the DB2_MDC_ROLLOUT registry variable. The value can be changed by invoking the SET CURRENT MDC ROLLOUT MODE statement. When the CURRENT MDC ROLLOUT MODE special register is set to a particular value, the execution behavior of subsequent DELETE statements that qualify for rollout is impacted. The DELETE statement does not need to be recompiled for the behavior to change.
CURRENT OPTIMIZATION PROFILE

The CURRENT OPTIMIZATION PROFILE special register specifies the qualified name of the optimization profile to be used by DML statements that are dynamically prepared for optimization.

The initial value is the null value. The value can be changed by invoking the SET CURRENT OPTIMIZATION PROFILE statement. An optimization profile that is not qualified with a schema name will be implicitly qualified with the value of the CURRENT DEFAULT SCHEMA special register.

Example 1: Set the optimization profile to 'JON.SALES'.

    SET CURRENT OPTIMIZATION PROFILE = JON.SALES

Example 2: Get the current value of the optimization profile name for this connection.

    VALUES (CURRENT OPTIMIZATION PROFILE) INTO :PROFILE
CURRENT PACKAGE PATH

The CURRENT PACKAGE PATH special register specifies a VARCHAR(4096) value that identifies the path to be used when resolving references to packages that are needed when executing SQL statements.

The value can be an empty or a blank string, or a list of one or more schema names that are delimited with double quotation marks and separated by commas. Any double quotation marks appearing as part of the string will need to be represented as two double quotation marks, as is common practice with delimited identifiers. The delimiters and commas contribute to the length of the special register.

This special register applies to both static and dynamic statements.

The initial value of CURRENT PACKAGE PATH in a user-defined function, method, or procedure is inherited from the invoking application. In other contexts, the initial value of CURRENT PACKAGE PATH is an empty string. The value is a list of schemas only if the application process has explicitly specified a list of schemas by means of the SET CURRENT PACKAGE PATH statement.

Examples:

An application will be using multiple SQLJ packages (in schemas SQLJ1 and SQLJ2) and a JDBC package (in schema DB2JAVA). Set the CURRENT PACKAGE PATH special register to check SQLJ1, SQLJ2, and DB2JAVA, in that order.

```
SET CURRENT PACKAGE PATH = "SQLJ1", "SQLJ2", "DB2JAVA"
```

Set the host variable HVPKLIST to the value currently stored in the CURRENT PACKAGE PATH special register.

```
VALUES CURRENT PACKAGE PATH INTO :HVPKLIST
```
The CURRENT PATH (or CURRENT_PATH) special register specifies a VARCHAR(2048) value that identifies the SQL path used when resolving unqualified function names, procedure names, data type names, global variable names, and module object names in dynamically prepared SQL statements.

CURRENT FUNCTION PATH is a synonym for CURRENT PATH. The initial value is the default value specified below. For static SQL, the FUNCPATH bind option provides an SQL path that is used for function and data type resolution.

The CURRENT PATH special register contains a list of one or more schema names that are enclosed by double quotation marks and separated by commas. For example, an SQL path specifying that the database manager is to look first in the FERMAT schema, then in the XGRAPHIC schema, and finally in the SYSIBM schema, is returned in the CURRENT PATH special register as:

"FERMAT", "XGRAPHIC", "SYSIBM"

The default value is “SYSIBM”, “SYSFUN”, “SYSPROC”, “SYSIBMADM”, “X”, where X is the value of the USER special register, delimited by double quotation marks. The value can be changed by invoking the SET CURRENT PATH statement. The schema SYSIBM does not need to be specified. If it is not included in the SQL path, it is implicitly assumed to be the first schema. SYSIBM does not take up any of the 2048 bytes if it is implicitly assumed.

A data type that is not qualified with a schema name will be implicitly qualified with the first schema in the SQL path that contains a data type with the same unqualified name. There are exceptions to this rule, as outlined in the descriptions of the following statements: CREATE TYPE (Distinct), CREATE FUNCTION, COMMENT, and DROP.

Example: Using the SYSCAT.ROUTINES catalog view, find all user-defined routines that can be invoked without qualifying the routine name, because the CURRENT PATH special register contains the schema name.

```
SELECT ROUTINENAME, ROUTINESCHEMA FROM SYSCAT.ROUTINES
    WHERE POSITION (ROUTINESCHEMA, CURRENT PATH, CODEUNITS16) <> 0
```
CURRENT QUERY OPTIMIZATION

The CURRENT QUERY OPTIMIZATION special register specifies an INTEGER value that controls the class of query optimization performed by the database manager when binding dynamic SQL statements. The QUERYOPT bind option controls the class of query optimization for static SQL statements. The possible values range from 0 to 9. For example, if the query optimization class is set to 0 (minimal optimization), then the value in the special register is 0. The default value is determined by the `dft_queryopt` database configuration parameter. The value can be changed by invoking the SET CURRENT QUERY OPTIMIZATION statement.

*Example:* Using the SYSCAT.PACKAGES catalog view, find all plans that were bound with the same setting as the current value of the CURRENT QUERY OPTIMIZATION special register.

```sql
SELECT PKGNAME, PKGSHEMA FROM SYSCAT.PACKAGES
WHERE QUERYOPT = CURRENT QUERY OPTIMIZATION
```
The CURRENT REFRESH AGE special register specifies a timestamp duration value with a data type of DECIMAL(20,6). It is the maximum duration since a particular timestamped event occurred to a cached data object (for example, a REFRESH TABLE statement processed on a system-maintained REFRESH DEFERRED materialized query table), such that the cached data object can be used to optimize the processing of a query. If CURRENT REFRESH AGE has a value of 99 999 999 999 999, and the query optimization class is 5 or more, the types of tables specified in CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION are considered when optimizing the processing of a dynamic SQL query.

The value of CURRENT REFRESH AGE must be 0 or 99 999 999 999 999. The initial value is 0. The value can be changed by invoking the SET CURRENT REFRESH AGE statement.
CURRENT SCHEMA

The CURRENT SCHEMA (or CURRENT_SCHEMA) special register specifies a VARCHAR(128) value that identifies the schema name used to qualify database object references, where applicable, in dynamically prepared SQL statements. For compatibility with DB2 for z/OS, CURRENT SQLID (or CURRENT_SQLID) can be specified in place of CURRENT SCHEMA.

The initial value of CURRENT SCHEMA is the authorization ID of the current session user. The value can be changed by invoking the SET SCHEMA statement.

The QUALIFIER bind option controls the schema name used to qualify database object references, where applicable, for static SQL statements.

Example: Set the schema for object qualification to 'D123'.

```
SET CURRENT SCHEMA = 'D123'
```
CURRENT SERVER

The CURRENT SERVER (or CURRENT_SERVER) special register specifies a VARCHAR(18) value that identifies the current application server. The register contains the actual name of the application server, not an alias.

CURRENT SERVER can be changed through the CONNECT statement, but only under certain conditions.

When used in an SQL statement inside a routine, CURRENT SERVER is not inherited from the invoking statement.

Example: Set the host variable APPL.Serve (VARCHAR(18)) to the name of the application server to which the application is connected.

VALUES CURRENT SERVER INTO :APPL.Serve
CURRENT SQL_CCFLAGS

The CURRENT SQL_CCFLAGS special register specifies the conditional compilation named constants that are defined for use during compilation of SQL statements.

The data type of the special register is VARCHAR(1024).

The CURRENT SQL_CCFLAGS special register contains a list of name and value pairs separated by a comma and a blank. The name is separated from the value in a pair using the colon character. The values in the list are a BOOLEAN constant, an INTEGER constant, or the keyword NULL. The names can be specified using any combination of uppercase or lowercase characters which are folded to all uppercase characters. For example, conditional compilation values defined for debug and tracing could appear in the special register as the string value:

```
CC_DEBUG:TRUE, CC_TRACE_LEVEL:2
```

The initial value of the special register is the value of the `sql_ccflags` database configuration parameter when the special register is first used. The first use can occur as a result of processing a statement with an inquiry directive or as a direct reference to the special register. If the value assigned to the `sql_ccflags` database configuration parameter is not valid, an error is returned on the first use (SQLSTATE 42815 or 428HV).

The value of the special register can be changed by executing the SET CURRENT SQL_CCFLAGS statement.
The CURRENT TIME (or CURRENT_TIME) special register specifies a time that is based on a reading of the time-of-day clock when the SQL statement is executed at the application server. If this special register is used more than once within a single SQL statement, or used with CURRENT DATE or CURRENT TIMESTAMP within a single statement, all values are based on a single clock reading.

When used in an SQL statement inside a routine, CURRENT TIME is not inherited from the invoking statement.

In a federated system, CURRENT TIME can be used in a query intended for data sources. When the query is processed, the time returned will be obtained from the CURRENT TIME register at the federated server, not from the data sources.

Example: Run the following command from the DB2 CLP to obtain the current time.

```
db2 values CURRENT TIME
```

Example: Using the CL_SCHED table, select all the classes (CLASS_CODE) that start (STARTING) later today. Today’s classes have a value of 3 in the DAY column.

```
SELECT CLASS_CODE FROM CL_SCHED
WHERE STARTING > CURRENT TIME AND DAY = 3
```
CURRENT TIMESTAMP

The CURRENT_TIMESTAMP (or CURRENT_TIMESTAMP) special register specifies a timestamp that is based on a reading of the time-of-day clock when the SQL statement is executed at the application server. If this special register is used more than once within a single SQL statement, or used with CURRENT DATE or CURRENT TIME within a single statement, all values are based on a single clock reading. It is possible for separate CURRENT_TIMESTAMP special register requests to return the same value; if unique values are required, consider using the GENERATE_UNIQUE function, a sequence, or an identity column.

If a timestamp with a specific precision is desired, the special register can be referenced as CURRENT_TIMESTAMP(integer), where integer can range from 0 to 12. The default precision is 6. The precision of the clock reading varies by platform and the resulting value is padded with zeros where the precision of the retrieved clock reading is less than the precision of the request.

When used in an SQL statement inside a routine, CURRENT_TIMESTAMP is not inherited from the invoking statement.

In a federated system, CURRENT_TIMESTAMP can be used in a query intended for data sources. When the query is processed, the timestamp returned will be obtained from the CURRENT_TIMESTAMP register at the federated server, not from the data sources.

SYSDATE can also be specified as a synonym for CURRENT_TIMESTAMP(0).

Example: Insert a row into the IN_TRAY table. The value of the RECEIVED column should be a timestamp that indicates when the row was inserted. The values for the other three columns come from the host variables SRC (char(8)), SUB (char(64)), and TXT (VARCHAR(200)).

```
INSERT INTO IN_TRAY
VALUES (CURRENT_TIMESTAMP, :SRC, :SUB, :TXT)
```
The CURRENT TIMEZONE special register specifies the difference between UTC (Coordinated Universal Time, formerly known as GMT) and local time at the application server. The difference is represented by a time duration (a decimal number in which the first two digits are the number of hours, the next two digits are the number of minutes, and the last two digits are the number of seconds). The number of hours is between -24 and 24 exclusive. Subtracting CURRENT TIMEZONE from a local time converts that local time to UTC. The time is calculated from the operating system time at the moment the SQL statement is executed. (The CURRENT TIMEZONE value is determined from C runtime functions.)

The CURRENT TIMEZONE special register can be used wherever an expression of the DECIMAL(6,0) data type is used; for example, in time and timestamp arithmetic.

When used in an SQL statement inside a routine, CURRENT TIMEZONE is not inherited from the invoking statement.

Example: Insert a record into the IN_TRAY table, using a UTC timestamp for the RECEIVED column.

```sql
INSERT INTO IN_TRAY VALUES (
    CURRENT_TIMESTAMP - CURRENT_TIMEZONE, 
    :source,
    :subject,
    :notetext )
```
The CURRENT USER (or CURRENT_USER) special register specifies the authorization ID that is to be used for statement authorization. For static SQL statements, the value represents the authorization ID that is used when the package is bound. For dynamic SQL statements, the value is the same as the value of the SESSION_USER special register for packages bound with the DYNAMICRULES(RUN) bind option. The data type of the register is VARCHAR(128).

Example: Select table names whose schema matches the value of the CURRENT USER special register.

```sql
SELECT TABNAME FROM SYSCAT.TABLES
WHERE TABSCHEMA = CURRENT USER AND TYPE = 'T'
```

If this statement is executed as a static SQL statement, it returns the tables whose schema name matches the binder of the package that includes the statement. If this statement is executed as a dynamic SQL statement, it returns the tables whose schema name matches the current value of the SESSION_USER special register.
SESSION_USER

The SESSION_USER special register specifies the authorization ID that is to be used for the current session. The value of this register is used for authorization checking of dynamic SQL statements when DYNAMICRULES run behavior is in effect for the package. The data type of the register is VARCHAR(128).

The initial value of SESSION_USER for a new connection is the same as the value of the SYSTEM_USER special register. Its value can be changed by invoking the SET SESSION AUTHORIZATION statement.

SESSION_USER is a synonym for the USER special register.

Example: Determine what routines can be executed using dynamic SQL. Assume DYNAMICRULES run behavior is in effect for the package that will issue the dynamic SQL statement that invokes the routine.

```
SELECT SCHEMA, SPECIFICNAME FROM SYSCAT.ROUTINEAUTH
WHERE GRANTEE = SESSION_USER
AND EXECUTEAUTH IN ('Y', 'G')
```
The SYSTEM_USER special register specifies the authorization ID of the user that connected to the database. The value of this register can only be changed by connecting as a user with a different authorization ID. The data type of the register is VARCHAR(128).

See “Example” in the description of the SET SESSION AUTHORIZATION statement.
The USER special register specifies the run-time authorization ID passed to the database manager when an application starts on a database. The data type of the register is VARCHAR(128).

When used in an SQL statement inside a routine, USER is not inherited from the invoking statement.

*Example:* Select all notes from the IN_TRAY table that were placed there by the user.

```
SELECT * FROM IN_TRAY
WHERE SOURCE = USER
```
Global variables are named memory variables that you can access and modify through SQL statements.

Global variables enable you to share relational data between SQL statements without the need for application logic to support this data transfer. You can control access to global variables through the GRANT (Global Variable Privileges) and REVOKE (Global Variable Privileges) statements.

DB2 supports created session global variables. A session global variable is associated with a specific session, and contains a value that is unique to that session. A created session global variable is available to any active SQL statement running against the database on which the variable was defined. A session global variable can be associated with more than one session, but its value will be specific to each session. Created session global variables and the privileges that are associated with them are defined in the system catalog.

User-defined session global variables are global variables that are created using an SQL data definition statement and registered to the database manager in the catalog. A global variable resides in the schema in which it was created or in the module where it was added or published. Schema variables are created using the CREATE VARIABLE statement. For more information, see “CREATE VARIABLE”. Module variables are created using the ADD module-variable-definition clause or PUBLISH module-variable-definition clause of the ALTER MODULE statement. For more information, see “ALTER MODULE”.

Resolving a global variable reference depends on the context where the global variable is referenced and how the global variable name is qualified. A variable reference that is intended to be a global variable could also resolve to an SQL variable, an SQL parameter, or a column name depending on the context of the reference and how the reference is qualified in that context. The following resolution steps assumes that the global variable reference does not resolve to an SQL variable, an SQL parameter, or a column name:

- If the global variable name is qualified, resolution is performed by the database manager using the following steps:
  1. If the global variable reference is from within a module and the qualifier matches the name of the module from within which the global variable is referenced, the module is searched for a matching module variable. If the qualifier is a single identifier, then the schema name of the module is ignored when matching the module name. If the qualifier is a two part identifier, then it is compared to the schema-qualified module name when determining a match. If a module variable matches the unqualified global variable name in the reference, resolution is complete. If the qualifier does not match or there is no matching module variable, then resolution continues with the next step.
  2. The qualifier is considered as a schema name and that schema is searched for a matching schema variable. If a schema variable matches the unqualified global variable name in the reference, resolution is complete. If the schema does not exist or there are no matching schema variables in the schema, and the qualifier matched the name of the module in the first step, then an error is returned (SQLSTATE 42703). Otherwise, resolution continues with the next step.
  3. The qualifier is considered as a module name.
Global variables

- If the module name is qualified with a schema name, then that module is searched for a matching published module variable.
- If the module name is not qualified with a schema name, then the schema for the module is the first schema in the SQL path that has a matching module name. If found, then that module is searched for a matching published module variable.
- If the module is not found using the SQL path, then the existence of a module public alias that matches the name of the global variable qualifier is considered. If found, then the module associated with the module public alias is searched for a matching published module variable.

If a published module variable matches the unqualified global variable name in the global variable reference, resolution is complete. If a matching module is not found or there is no matching module variable in the matching module, an error is returned (SQLSTATE 42703).

- If the global variable name is unqualified, resolution is performed by the database manager using the following steps:
  1. If an unqualified global variable reference is from within a module object, the module is searched for a matching module variable. If a module variable matches the global variable name in the reference, resolution is complete. If there is no matching module variable, then resolution continues with the next step.
  2. The schemas in the SQL path are searched in order from left to right for a matching schema variable. If a schema variable matches the global variable name in the reference, resolution is complete.

If no matching global variable is found after completing step 2, an error is returned (SQLSTATE 42703).

When a global variable is referenced within an SQL statement or within a trigger, view, or routine, a dependency on the fully qualified global variable name is recorded for the statement or object. The authorization required for a global variable depends on where it is defined and how it is used.

- The authorization ID of an SQL statement which references a schema variable and retrieves the value must have the READ privilege on the global variable.
- The authorization ID of an SQL statement which references a schema variable and assigns a value must have the WRITE privilege on the global variable.
- The authorization ID of an SQL statement which references a module variable and either retrieves the value or assigns a value must have the EXECUTE privilege on the module of the global variable.

Global variables can be referenced within any expression that does not need to be deterministic. Deterministic expressions are required in the following situations, which preclude the use of global variables:

- Check constraints
- Definitions of generated columns
- Refresh immediate materialized query tables (MQTs)

The value of a global variable can be changed using the EXECUTE, FETCH, SET, SELECT INTO, or VALUES INTO statement. It can also be changed if it is an argument of an OUT or INOUT parameter in a CALL statement or function invocation.
Global variables

The following table shows at what point the value of a global variable is read for the indicated reference of the global variable.

Table 19. When the value of a global variable is read based on the context

<table>
<thead>
<tr>
<th>Context of a global variable reference:</th>
<th>The reference uses the value of the global variable at the beginning of:</th>
</tr>
</thead>
<tbody>
<tr>
<td>A statement in a compound SQL (inlined) statement</td>
<td>The compound SQL (inlined) statement</td>
</tr>
<tr>
<td>A statement in a compound SQL (compiled) statement</td>
<td>The statement in the compound SQL (compiled) statement</td>
</tr>
<tr>
<td>A statement, that possibly invokes a function or activates a trigger¹</td>
<td>The SQL statement</td>
</tr>
<tr>
<td>A statement in an invoked inlined SQL function</td>
<td>The SQL statement invoking the inlined SQL function</td>
</tr>
<tr>
<td>A statement in an activated inlined trigger</td>
<td>The SQL statement activating the inlined trigger</td>
</tr>
<tr>
<td>A statement in an invoked inlined SQL method</td>
<td>The SQL statement invoking the SQL method</td>
</tr>
<tr>
<td>A statement in an invoked compiled SQL function</td>
<td>The SQL statement in the compiled SQL function</td>
</tr>
<tr>
<td>A statement in an activated compiled trigger</td>
<td>The SQL statement in the compiled trigger</td>
</tr>
<tr>
<td>A statement in an invoked external routine</td>
<td>The SQL statement in the external program</td>
</tr>
</tbody>
</table>

Note: In this table, the SQL statement, which might invoke a function or activates a trigger, does not include compound SQL (inlined) statement and compound SQL (compiled) statement.

If the data type of the global variable is a cursor type, the underlying cursor of the global cursor variable can be referenced anywhere that a cursor-variable-name can be specified.

If the data type of the global variable is a row type, a field of the global row variable can be referenced anywhere that a global variable with the same type as the field can be referenced. The global variable name that qualifies the field name is resolved in the same way as any other global variable.
Functions

A function is an operation denoted by a function name followed by one or more operands that are enclosed in parentheses. For example, the TIMESTAMP function can be passed input data values of type DATE and TIME, and the result is a TIMESTAMP. Functions can be either built-in or user-defined.

- **Built-in functions** are provided with the database manager. They return a single result value and are identified as part of the SYSIBM schema. Such functions include aggregate functions (for example, AVG), operator functions (for example, +), casting functions (for example, DECIMAL), and scalar functions (for example, CEILING).

- **User-defined functions** are functions that are created using an SQL data definition statement and registered to the database manager in the catalog. User-defined schema functions are created using the CREATE FUNCTION statement. For more information, see “CREATE FUNCTION”. A set of user-defined schema functions is provided with the database manager in a schema called SYSFUN. User-defined module functions are created using the ALTER MODULE ADD FUNCTION or ALTER MODULE PUBLISH FUNCTION statements. For more information, see “ALTER MODULE”. A set of user-defined module functions is provided with the database manager in a set of modules in a schema called SYSIBMADM. A user-defined function resides in the schema in which it was created or in the module where it was added or published.

User-defined functions extend the capabilities of the database system by adding function definitions (provided by users or third party vendors) that can be applied in the database engine itself. Extending database functions lets the database exploit the same functions in the engine that an application uses, providing more synergy between application and database.

**External, SQL, and sourced user-defined functions**

A user-defined function can be an external function, an SQL function, or a sourced function. An external function is defined to the database with a reference to an object code library, and a function within that library that will be executed when the function is invoked. External functions cannot be aggregate functions. An SQL function is defined to the database using only SQL statements, including at least one RETURN statement. It can return a scalar value, a row, or a table. SQL functions cannot be aggregate functions. A sourced function is defined to the database with a reference to another built-in or user-defined function that is already known to the database. Sourced functions can be scalar functions or aggregate functions. They are useful for supporting existing functions with user-defined types.

**Scalar, aggregate, row, and table user-defined functions**

Each user-defined function is also categorized as a scalar, aggregate, row, or table function. A scalar function is a function that returns a single-valued answer each time it is called. For example, the built-in function SUBSTR() is a scalar function. Scalar UDFs can be either external or sourced.

An aggregate function is one which conceptually is passed a set of like values (a column) and returns a single-valued answer. An example of an aggregate function is the built-in function AVG(). An external column UDF cannot be defined to DB2, but a column UDF, which is sourced upon one of the built-in aggregate functions, can be defined. This is useful for distinct types. For example, if there is a distinct
Functions

type SHOESIZE defined with base type INTEGER, a UDF AVG(SHOESIZE), which is sourced on the built-in function AVG(INTEGER), could be defined, and it would be an aggregate function.

A row function is a function that returns one row of values. It can be used in a context where a row expression is supported. It can also be used as a transform function, mapping attribute values of a structured type into values in a row. A row function must be defined as an SQL function.

A table function is a function that returns a table to the SQL statement which references it. It may only be referenced in the FROM clause of a SELECT statement. Such a function can be used to apply SQL language processing power to data that is not DB2 data, or to convert such data into a DB2 table. A table function can read a file, get data from the Web, or access a Lotus Notes® database and return a result table. This information can be joined with other tables in the database. A table function can be defined as an external function or as an SQL function. (A table function cannot be a sourced function.)

Function signatures

A schema function is identified by its schema name, a function name, the number of parameters, and the data types of its parameters. A module function is identified by its schema name, module name, a function name, the number of parameters, and the data types of its parameters. This identification of a schema function or a module function is called a function signature, which must be unique within the database; for example, TEST.RISK(INTEGER). There can be more than one function with the same name in a schema or a module, provided that the number of parameters or the data types of the parameters are different. A function name for which there are multiple function instances with the same number of parameters is called an overloaded function. A function name can be overloaded within a schema, in which case there is more than one function by that name with the same number of parameters in the schema. Similarly, a function name can be overloaded within a module, in which case there is more than one function by that name with the same number of parameters in the module. These functions must have different parameter data types. Functions can also be overloaded across the schemas of an SQL path, in which case there is more than one function by that name with the same number of parameters in different schemas of the SQL path. These functions do not necessarily have different parameter data types.

Function invocation

Each reference to a function conforms to the following syntax:

```
function-name (expression) ALL, DISTINCT
```

Notes:

1. The ALL or DISTINCT keyword can be specified only for an aggregate function or a user-defined function that is sourced on an aggregate function.
In the above syntax, *expression* and *row-expression* cannot include an aggregate function. See “Expressions” for other rules for *expression*.

A function is invoked by referring (in an allowable context) to its qualified or unqualified function name followed by the list of arguments enclosed in parentheses. The possible qualifiers for a function name are:

- A schema name
- An unqualified module name
- A schema-qualified module name

The qualifier used when invoking a function determines the scope used to search for a matching function.

- If a schema-qualified module name is used as the qualifier, the scope is the specified module.
- If a single identifier is used as the qualifier, the scope includes:
  - The schema that matches the qualifier
  - One of the following modules:
    - The invoking module, if the invoking module name matches the qualifier
    - The first module in a schema in the SQL path that matches the qualifier
- If no qualifier is used, the scope includes the schemas in the SQL path and, if the function is invoked from within a module object, the same module from which the function is invoked.

For static SQL statements, the SQL path is specified using the **FUNCPATH** bind option. For dynamic SQL statements, the SQL path is the value of the CURRENT PATH special register.

When any function is invoked, the database manager must determine which function to execute. This process is called **function resolution** and applies to both built-in and user-defined functions. It is recommended that function invocations intending to invoke a user-defined function be fully qualified. This improves performance of function resolution and prevents unexpected function resolution results as new functions are added or privileges granted.

An *argument* is a value passed to a function upon invocation. When a function is invoked in SQL, it is passed a list of zero or more arguments. They are positional in that the semantics of an argument are determined by its position in the argument list. A *parameter* is a formal definition of an input to a function or an output from a function. When a function is defined to the database, either internally (a built-in function) or by a user (a user-defined function), its parameters (zero or more) are specified, and the order of their definitions defines their positions and their semantics. Therefore, every parameter is a particular positional input to a function or an output from a function. On invocation, an argument corresponds to a particular parameter by virtue of its position in the list of arguments. In cases where the arguments of the function invocation are not an exact match to the data types of the parameters of the selected function, the arguments are converted to the data type of the parameter at execution using the same rules as assignment to columns. This includes the case where precision, scale, or length differs between the argument and the parameter.

Access to schema functions is controlled through the EXECUTE privilege on the schema functions. If the authorization ID of the statement invoking the function does not have EXECUTE privilege, the schema function will not be considered by
the function resolution algorithm, even if it is a better match. Built-in functions (SYSIBM functions) and functions in the SYSFUN schema have the EXECUTE privilege implicitly granted to PUBLIC.

Access to module functions is controlled through EXECUTE privilege on the module for all functions within the module. The authorization ID of the statement invoking the function might not have EXECUTE privilege on a module. In such cases, module functions within that module, unlike schema functions, are still considered by the function resolution algorithm even though they cannot be executed.

When the user-defined function is invoked, the value of each of its arguments is assigned, using storage assignment, to the corresponding parameter of the function. Control is passed to external functions according to the calling conventions of the host language. When execution of a user-defined scalar function or a user-defined aggregate function is complete, the result of the function is assigned, using storage assignment, to the result data type. For details on the assignment rules, see “Assignments and comparisons”.

Table functions can be referenced only in the FROM clause of a subselect. For more details on referencing a table function, see “table-reference”.

## Function resolution

After a function is invoked, the database manager must determine which function to execute. This process is called function resolution and applies for both built-in and user-defined functions.

Function resolution has two steps.

1. In the first step, the database manager determines the set of candidate functions based on the qualification of the name of the invoked function, the context that invokes the function, the unqualified name of the invoked function, and the number of arguments specified.

2. In the second step, the database manager determines the best fit from the set of candidate functions based on the data types of the arguments of the invoked function as compared with the data types of the parameters of the functions in the set of candidate functions.

## Determining the set of candidate functions

The functions selected for the set of candidate functions are from one or more of the following search spaces.

1. The context module, that is, the module which contains the module object that invoked the function

2. A set of one or more schemas

3. A module other than the context module

The specific search spaces considered are affected by the qualification of the name of the invoked function.

- **Qualified function resolution**: When a function is invoked with a function name and a qualifier, the database manager uses the qualifier and, in some cases, the context of the invoked function to determine the set of candidate functions.

  1. If a function is invoked from within a module object using a function name with a qualifier, the database manager considers if the qualifier matches the context module name. If the qualifier is a single identifier, then the schema
name of the module is ignored when determining a match. If the qualifier is a two-part identifier, then it is compared to the schema-qualified module name when determining a match. If the qualifier matches the context module name, the database manager searches the context module for candidate functions based on the following criteria:

– The name of the function instance must match the name in the function invocation.
– The number of parameters in the function instance must match the number of arguments in the function invocation.

If one or more candidate functions are found in the context module, then this set of candidate functions is processed for best fit without consideration of possible candidate functions in any other search space (see “Determining the best fit”). Otherwise, continue to the next search space.

2. If the qualifier is a single identifier, the database manager considers the qualifier as a schema name and searches that schema for candidate functions based on the following criteria:

– The name of the function instance must match the name in the function invocation.
– The number of parameters in the function instance must match the number of arguments in the function invocation.
– The authorization ID of the statement must have the EXECUTE privilege to the function instance.

If one or more candidate functions are found in the schema, then this set of candidate functions is processed for best fit without consideration of possible candidate functions in any other search space (see “Determining the best fit”). Otherwise, continue to the next search space, if applicable.

3. If the function is invoked from outside a module or the qualifier does not match the context module name when it is invoked from within a module object, the database manager considers the qualifier as a module name. Without considering EXECUTE privilege on modules, the database manager then selects the first module that matches based on the following criteria:

– If the module name is qualified with a schema name, select the module with that schema name and module name.
– If the module name is not qualified with a schema name, select the module with that module name that is found in the schema earliest in the SQL path.
– If the module is not found using the SQL path, select the module public alias with that module name.

If a matching module is not found, then there are no candidate functions. If a matching module is found, the database manager searches the selected module for candidate functions based on the following criteria:

– The name of the function instance must match the name in the function invocation.
– The number of parameters in the function instance must match the number of arguments in the function invocation.
– The function must be a published module function.

If one or more candidate functions are found in the selected modules, then this set of candidate functions is processed for best fit (see “Determining the best fit”).
If the database manager does not find any candidate, an error is returned (SQLSTATE 42884).

- **Unqualified function resolution**: When a function is invoked without a qualifier, the database manager considers the context of the invoked function to determine the sets of candidate functions.

  1. If a function is invoked with an unqualified function name from within a module object, the database manager searches the context module for candidate functions based on the following criteria:
     - The name of the function instance must match the name in the function invocation.
     - The number of parameters in the function instance must match the number of arguments in the function invocation.

     If one or more candidate functions are found in the context module, then these candidate functions are included with any candidate functions from the schemas in the SQL path (see next item).

  2. If a function is invoked with an unqualified function name, either from within a module object or from outside a module, the database manager searches the list of schemas in the SQL path to resolve the function instance to execute. For each schema in the SQL path (see “SQL path”), the database manager searches the schema for candidate functions based on the following criteria:
     - The name of the function instance must match the name in the function invocation.
     - The number of parameters in the function instance must match the number of function arguments in the function invocation.
     - The authorization ID of the statement must have the EXECUTE privilege to the function instance.

     If one or more candidate functions are found in the schemas in the SQL path, then these candidate functions are included with any candidate functions from the context module (see previous item). This set of candidate functions is processed for best fit (see ”Determining the best fit”).

If the database manager does not find any candidate functions, an error is returned (SQLSTATE 42884).

**Determining the best fit**

The set of candidate functions may contain one function, or more than one function with the same name. In either case, the data types of the parameters of each function in the set of candidate functions are used to determine if the function meets the best fit requirements.

The database manager determines the function, or set of functions, that meet the best fit requirements for the invocation by comparing the argument and parameter data types. Note that the data type of the result of the function or the type of function (aggregate, scalar, or table) under consideration does not enter into this determination.

When determining whether the data types of the parameters are the same as the arguments:
- Synonyms of data types match. For example, FLOAT and DOUBLE are considered to be the same.
Functions

- Attributes of a data type such as length, precision, scale, and code page are ignored. Therefore, CHAR(8) and CHAR(35) are considered to be the same, as are DECIMAL(11,2), and DECIMAL(4,3).

A subset of the candidate functions is obtained by considering only those functions for which the data type of each argument of the function invocation matches or is promotable to the data type of the corresponding parameter of the function instance. The precedence list for the promotion of data types in “Promotion of data types” shows the data types that fit (considering promotion) for each data type in best-to-worst order. If this subset is not empty, then the best fit is determined using the Promotable process on this subset of candidate functions. If this subset is empty, then the best fit is determined using the Castable process on the original set of candidate functions.

Promotable process
This process determines the best fit only considering whether arguments in the function invocation match or can be promoted to the data type of the corresponding parameter of the function definition. For the subset of candidate functions, compare the parameter lists from left to right, using the following process:

1. Compare the data type of the argument in the function invocation to the data type of the corresponding parameter in each candidate function definition. (As stated previously, synonyms of data types match and attributes of data type are ignored.)

2. For this argument, if one candidate function has a data type that fits (only considering promotion) the function invocation better than the data types in the other candidate functions, that function is the best fit. The precedence list for the promotion of data types in “Promotion of data types” shows the data types that fit (considering promotion) for each data type in best-to-worst order.

3. If the data type of the parameter for more than one candidate function fits (only considering promotion) the function invocation equally well, eliminate those candidate functions that do not fit the function invocation equally well. Repeat this process for the next argument of the function invocation.

If only one candidate function remains after comparing all arguments, that function is the best fit. If more than one candidate function remains, then all the candidate functions that remain are equally the best fit. In this case, the database manager selects the function from the context module, if still a candidate function, and otherwise selects the function whose schema is earliest in the SQL path.

If a function is selected, its successful use depends on it being invoked in a context in which the returned result is allowed. For example, if the function returns a table where a table is not allowed, an error is returned.

Castable process
This process determines the best fit first considering if the arguments in the function invocation match or can be promoted to the data type of the corresponding parameter of the function definition and then if the arguments can be implicitly cast for function resolution to the data type of the corresponding parameter. For the set of candidate functions, compare the parameter lists from left to right, using the following process:

1. Compare the data type of the argument in the function invocation to the data type of the corresponding parameter in each candidate
function definition. (As stated previously, synonyms of data types
match and attributes of data type are ignored.)

2. For this argument, if one candidate function has a data type that fits
(only considering promotion) the function invocation better than the
data types in the other candidate functions, that function is the best fit.
The precedence list for the promotion of data types in “Promotion of
data types” shows the data types that fit (considering promotion) for
each data type in best-to-worst order. In this case, where there is only
one candidate function remaining, the remaining arguments are
evaluated to ensure that each argument is promotable or castable to the
corresponding parameter. If not, an error is returned (SQLSTATE
42884).

3. When the data type of the parameter for more than one candidate
function fits (only considering promotion) the function invocation
equally well, eliminate those candidate functions that do not fit the
function invocation equally well. If the data type of the parameter does
not fit (only considering promotion) the function invocation for any
candidate function, no candidate functions are eliminated. Repeat this
process for the next argument of the function invocation.

If more than one candidate function remains after processing the
arguments, compare each parameter that had a data type that did not fit
(only considering promotion) from left to right, using the following
process:

1. Check the data types of the corresponding parameters for all remaining
candidate functions. If all the data types do not belong to the same data
type precedence list, as specified in “Promotion of data types”, an error
is returned (SQLSTATE 428F5).

2. If the data type of the argument cannot be implicitly cast to the data
type of the corresponding parameter, as specified in Implicit casting for
function resolution, an error is returned (SQLSTATE 42884).

3. Compare the data type of the argument in the function invocation to
the data type of the parameter in each candidate function. If one
candidate function has a data type that fits (considering implicit
casting) the function invocation better than the data types in the other
candidate functions, that function is the best fit. The data type list in
“Implicit casting for function resolution” shows the data type that fits
(considering implicit casting) better. Eliminate those candidate functions
that do not fit the function invocation equally well (considering implicit
casting). Repeat this process for the next argument of the function
invocation that had a data type that did not fit (only considering
promotion).

If only one candidate function remains after comparing all arguments, that
function is the best fit. If more than one candidate function remains, then
all the candidate functions that remain are equally the best fit. In this case,
the database manager selects the function from the context module, if still
a candidate function, and otherwise selects the function whose schema is
earliest in the SQL path.

Once a function is selected, it is still possible that an error may be
returned.

• If a module function is selected and either the function is invoked from
outside a module or the function is invoked from within a module object
and the qualifier does not match the context module name, the

Chapter 2. Language elements  197
authorization ID of the statement that invoked the function must have EXECUTE privilege on the module that contains the selected function (SQLSTATE 42501).

- If a function is selected, its successful use depends on it being invoked in a context in which the returned result is allowed. For example, if the function returns a table where a table is not allowed, an error is returned (SQLSTATE 42887).
- If a cast function, either built-in or user-defined, and any argument would need to be implicitly cast (not promoted) to the data type of the parameter, an error is returned (SQLSTATE 42884).
- If a function invocation involves an argument with an unnamed row type, an error is returned (SQLSTATE 42884) if either of the following conditions occur:
  - The number of fields of the argument’s unnamed row type does not match the number of fields of the parameter’s unnamed row type
  - The data types of the argument’s fields are not assignable to the corresponding data type of the parameter’s fields

**Implicit casting for function resolution**

Implicit casting for function resolution is not supported for arguments with a user-defined type, reference type, or XML data type. It is also not supported for built-in or user-defined cast functions. It is supported for the following cases:

- A value of one data type can be cast to any other data type that is in the same data type precedence list, as specified in “Promotion of data types”.
- A numeric or datetime data type can be cast to a character or graphic string data type, except for LOBs
- A character or graphic string type, except LOBs, can be cast to a numeric or datetime data type
- A character FOR BIT DATA can be cast to a BLOB and a BLOB can be cast to a character FOR BIT DATA
- A TIMESTAMP data type can be cast to a TIME data type
- An untyped argument can be cast to a string, numeric, or datetime data type.

Similar to the data type precedence list for promotion, for implicit casting there is an order to the data types that are in the group of related data types. This order is used when performing function resolution that considers implicit casting. Table 20 shows the data type ordering for implicit casting for function resolution. The data types are listed in best-to-worst order (note that this is different than the ordering in the data type precedence list for promotion). Note, when function resolution selects a built-in function and implicit casting is necessary for some argument, if the built-in function supports both character input and graphic input for the parameter, the argument is implicitly cast to character.

**Table 20. Data type ordering for implicit casting for function resolution**

<table>
<thead>
<tr>
<th>Data type group</th>
<th>Data type list for implicit casting for function resolution (in best-to-worst order)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Numeric data types</td>
<td>DECIMAL, DECIMAL(38), NUMERIC, REAL, SMALLINT, BIGINT, INTEGER, DECIMAL</td>
</tr>
</tbody>
</table>

SQL Reference, Volume 1
Table 20. Data type ordering for implicit casting for function resolution (continued)

<table>
<thead>
<tr>
<th>Data type group</th>
<th>Data type list for implicit casting for function resolution (in best-to-worst order)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Character and graphic string data types</td>
<td>VARCHAR or VARGRAPHIC, CHAR or GRAPHIC, CLOB or DBCLOB</td>
</tr>
<tr>
<td>Datetime data types</td>
<td>TIMESTAMP, DATE</td>
</tr>
</tbody>
</table>

Notes:
1. The lower case types above are defined as follows:
   - decimal = DECIMAL \((p,s)\) or NUMERIC\((p,s)\)
   - real = REAL or FLOAT\(n\) where \(n\) is not greater than 24
   - double = DOUBLE, DOUBLE-PRECISION, FLOAT or FLOAT\(n\), where \(n\) is greater than 24

   Shorter and longer form synonyms of the listed data types are considered to be the same as the listed form.
2. For a Unicode database only, the following are considered to be equivalent data types:
   - CHAR or GRAPHIC
   - VARCHAR and VARGRAPHIC
   - CLOB and DBCLOB

Table 21. Derived length of an argument when invoking a built-in scalar function in cases where implicit casting is needed.

<table>
<thead>
<tr>
<th>Source Data Type</th>
<th>Char</th>
<th>Graphic</th>
<th>Varchar</th>
<th>Vargraphic</th>
<th>Clob</th>
<th>DBclob</th>
<th>Blob</th>
<th>Timestamp</th>
<th>Decfloat</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNTYPED</td>
<td>127</td>
<td>127</td>
<td>254</td>
<td>254</td>
<td>32767</td>
<td>32767</td>
<td>32767</td>
<td>12</td>
<td>34</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>INTEGER</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>BIGINT</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DECIMAL((p,s))</td>
<td>2(p)</td>
<td>2(p)</td>
<td>2(p)</td>
<td>2(p)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>REAL</td>
<td>24</td>
<td>24</td>
<td>24</td>
<td>24</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>24</td>
<td>24</td>
<td>24</td>
<td>24</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DECFLOAT</td>
<td>42</td>
<td>42</td>
<td>42</td>
<td>42</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>CHAR(n)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>min(n,254)</td>
<td>12</td>
</tr>
<tr>
<td>VARCHAR(n)</td>
<td>min(n,254)</td>
<td>min(n,127)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>min(n,32672)</td>
<td>12</td>
<td>34</td>
</tr>
<tr>
<td>CLOB(n)</td>
<td>min(n,254)</td>
<td>min(n,127)</td>
<td>min(n,32672)</td>
<td>min(n,16336)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>GRAPHIC(n)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>VARGRAPHIC(n)</td>
<td>min(n,254)</td>
<td>min(n,127)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>12</td>
<td>34</td>
</tr>
<tr>
<td>DBCLOB(n)</td>
<td>min(n,254)</td>
<td>min(n,127)</td>
<td>min(n,32672)</td>
<td>min(n,16336)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>BLOB(n)</td>
<td>min(n,254)</td>
<td>-</td>
<td>min(n,32672)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>TIME</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DATE</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>TIMESTAMP((p))</td>
<td>if (p=0) then</td>
<td>if (p=0) then</td>
<td>if (p=0) then</td>
<td>if (p=0) then</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>19 else (p+20)</td>
<td>19 else (p+20)</td>
<td>19 else (p+20)</td>
<td>19 else (p+20)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

**SQL path considerations for built-in functions**

Built-in functions reside in a special schema called SYSIBM. Additional functions are available in the SYSFUN, SYSPROC, and SYSIBMADM schemas as well as in modules within the SYSIBMADM schema, but are not considered built-in functions.
Functions

because they are developed as user-defined functions and have no special processing considerations. Users cannot define additional functions in the SYSIBM, SYSFUN, SYSPROC, or SYSIBMADM schemas (or in any other schema whose name begins with the letters 'SYS', except SYSTOOLS).

As already stated, the built-in functions participate in the function resolution process exactly as do the user-defined functions. One difference between built-in and user-defined functions, from a function resolution perspective, is that the built-in functions must always be considered during function resolution. Therefore, omission of SYSIBM from the path results in the assumption (for function and data type resolution) that SYSIBM is the first schema on the path.

For example, if a user's SQL path is defined as:
"SHAREFUN","SYSIBM","SYSFUN"

and there is a LENGTH function defined in schema SHAREFUN with the same number and types of arguments as SYSIBM.LENGTH, then an unqualified reference to LENGTH in this user's SQL statement will result in selecting SHAREFUN.LENGTH. However, if the user's SQL path is defined as:
"SHAREFUN","SYSFUN"

and the same SHAREFUN.LENGTH function exists, then an unqualified reference to LENGTH in this user's SQL statement will result in selecting SYSIBM.LENGTH, because SYSIBM implicitly appears first in the path.

To minimize potential problems in this area:

• Never use the names of built-in functions for user-defined functions.
• If, for some reason, it is necessary to create a user-defined function with the same name as a built-in function, be sure to qualify any references to it.

Note: Some invocations of built-in functions do not support SYSIBM as an explicit qualifier and resolve directly to the built-in function without considering the SQL path. Specific cases are covered in the description of the built-in function.

Examples of function resolution

The following are examples of function resolution. (Note that not all required keywords are shown.)

• This is an example illustrating the SQL path considerations in function resolution. For this example, there are eight ACT functions, in three different schemas, registered as:

```
CREATE FUNCTION AUGUSTUS.ACT (CHAR(5), INT, DOUBLE) SPECIFIC ACT_1 ...  
CREATE FUNCTION AUGUSTUS.ACT (INT, INT, DOUBLE) SPECIFIC ACT_2 ...  
CREATE FUNCTION AUGUSTUS.ACT (INT, INT, DOUBLE, INT) SPECIFIC ACT_3 ...  
CREATE FUNCTION JULIUS.ACT (INT, DOUBLE, DOUBLE) SPECIFIC ACT_4 ...  
CREATE FUNCTION JULIUS.ACT (INT, INT, DOUBLE) SPECIFIC ACT_5 ...  
CREATE FUNCTION JULIUS.ACT (SMALLINT, INT, DOUBLE) SPECIFIC ACT_6 ...  
CREATE FUNCTION JULIUS.ACT (INT, INT, DECFLOAT) SPECIFIC ACT_7 ...  
CREATE FUNCTION NERO.ACT (INT, INT, DEC(7,2)) SPECIFIC ACT_8 ...  
```

The function reference is as follows (where I1 and I2 are INTEGER columns, and D is a DECIMAL column):

```
SELECT ... ACT(I1, I2, D) ...  
```

Assume that the application making this reference has an SQL path established as:
"JULIUS","AUGUSTUS","CAESAR"
Following through the algorithm...

- The function with specific name ACT_8 is eliminated as a candidate, because the schema NERO is not included in the SQL path.

- The function with specific name ACT_3 is eliminated as a candidate, because it has the wrong number of parameters. ACT_1 and ACT_6 are eliminated because, in both cases, the first argument cannot be promoted to the data type of the first parameter.

- Because there is more than one candidate remaining, the arguments are considered in order.

- For the first argument, the remaining functions, ACT_2, ACT_4, ACT_5, and ACT_7 are an exact match with the argument type. No functions can be eliminated from consideration; therefore the next argument must be examined.

- For this second argument, ACT_2, ACT_5, and ACT_7 are exact matches, but ACT_4 is not, so it is eliminated from consideration. The next argument is examined to determine some differentiation among ACT_2, ACT_5, and ACT_7.

- For the third and last argument, neither ACT_2, ACT_5, nor ACT_7 match the argument type exactly. Although ACT_2 and ACT_5 are equally good, ACT_7 is not as good as the other two because the type DOUBLE is closer to DECIMAL than is DECIMAL. ACT_7 is eliminated.

- There are two functions remaining, ACT_2 and ACT_5, with identical parameter signatures. The final tie-breaker is to see which function’s schema comes first in the SQL path, and on this basis, ACT_5 is the function chosen.

This is an example of a situation where function resolution will result in an error (SQLSTATE 428F5) since more than one candidate function fits the invocation equally well, but the corresponding parameters for one of the arguments do not belong to the same type precedence list.

For this example, there are only three function in a single schema defined as follows:

```
CREATE FUNCTION CAESAR.ACT (INT, VARCHAR(5), VARCHAR(5)) SPECIFIC ACT_1 ...
CREATE FUNCTION CAESAR.ACT (INT, INT, DATE)    SPECIFIC ACT_2 ...
CREATE FUNCTION CAESAR.ACT (INT, INT, DOUBLE)  SPECIFIC ACT_3 ...
```

The function reference is as follows (where I1 and I2 are INTEGER columns, and VC is a VARCHAR column):

```
SELECT ... ACT(I1, I2, VC) ...
```

Assume that the application making this reference has an SQL path established as:

```
"CAESAR"
```

Following through the algorithm ...

- Each of the candidate functions is evaluated to determine if the data type of each input argument of the function invocation matches or is promotable to the data type of the corresponding parameter of the function instance:
  - For the first argument, all the candidate functions have an exact match with the parameter type.
  - For the second argument, ACT_1 is eliminated because INTEGER is not promotable to VARCHAR.
  - For the third argument, both ACT_2 and ACT_3 are eliminated since VARCHAR is not promotable to DATE or DOUBLE, so no candidate functions remain.

- Since the subset of candidate functions from above is empty, the candidate functions are considered using the castable process:
Functions

- For the first argument, all the candidate functions have an exact match with
the parameter type.
- For the second argument, ACT_1 is eliminated since INTEGER is not
promotable to VARCHAR. ACT_2 and ACT_3 are better candidates.
- For the third argument, the data type of the corresponding parameters of
ACT_2 and ACT_3 do not belong to the same data type precedence list, so
an error is returned (SQLSTATE 428F5).

This example illustrates a situation where function resolution will succeed using
the castable process. For this example, there are only three function in a single
schema defined as follows:

CREATE FUNCTION CAESAR.ACT (INT, VARCHAR(5), VARCHAR(5)) SPECIFIC ACT_1 ...
CREATE FUNCTION CAESAR.ACT (INT, INT, DECFLOAT) SPECIFIC ACT_2 ...
CREATE FUNCTION CAESAR.ACT (INT, INT, DOUBLE) SPECIFIC ACT_3 ...

The function reference is as follows (where I1 and I2 are INTEGER columns, and
VC is a VARCHAR column):

SELECT ... ACT(I1, I2, VC) ...

Assume that the application making this reference has an SQL path established
as:
"CAESAR"

Following through the algorithm ...
- Each of the candidate functions is evaluated to determine if the data type of
each input argument of the function invocation matches or is promotable to
the data type of the corresponding parameter of the function instance:
- For the first argument, all the candidate functions have an exact match with
the parameter type.
- For the second argument, ACT_1 is eliminated because INTEGER is not
promotable to VARCHAR.
- For the third argument, both ACT_2 and ACT_3 are eliminated since
VARCHAR is not promotable to DECFLOAT or DOUBLE, so no candidate
functions remain.
- Since the subset of candidate functions from above is empty, the candidate
functions are considered using the castable process:
- For the first argument, all the candidate functions have an exact match with
the parameter type.
- For the second argument, ACT_1 is eliminated since INTEGER is not
promotable to VARCHAR. ACT_2 and ACT_3 are better candidates.
- For the third argument, both DECIMAL and DOUBLE are in the same
data type precedence list and VARCHAR can be implicitly cast to both
DECIMAL and DOUBLE. Since DECIMAL is a better fit for the purpose
of implicit casting, ACT_2 is the best fit

This example illustrates that during function resolution using the castable
process that promotion of later arguments takes precedence over implicit casting.
For this example, there are only three function in a single schema defined as
follows:

CREATE FUNCTION CAESAR.ACT (INT, INT, VARCHAR(5)) SPECIFIC ACT_1 ...
CREATE FUNCTION CAESAR.ACT (INT, INT, DECIMAL) SPECIFIC ACT_2 ...
CREATE FUNCTION CAESAR.ACT (INT, INT, DOUBLE) SPECIFIC ACT_3 ...

The function reference is as follows (where I1 is an INTEGER column, and VC1
is a VARCHAR column and C1 is a CHAR column):

SELECT ... ACT(I1, VC1, C1) ...

Assume that the application making this reference has an SQL path established
as:
Following through the algorithm:

- Each of the candidate functions is evaluated to determine if the data type of each input argument of the function invocation matches or is promotable to the data type of the corresponding parameter of the function instance:
  - For the first argument, all the candidate functions have an exact match with the parameter type.
  - For the second argument, all candidate functions are eliminated since VARCHAR is not promotable to INTEGER, so no candidate functions remain.
  - Since the subset of candidate functions from above is empty, the candidate functions are considered using the castable process
    - For the first argument, all the candidate functions have an exact match with the parameter type.
    - For the second argument, none of the candidate functions have a parameter to which the corresponding argument can be promoted, so no candidate functions are eliminated.
    - Since the third argument can be promoted to the parameter of ACT_1, but not to the parameters of ACT_2 or ACT_3, ACT_1 is the best fit.
Methods

A database method of a structured type is a relationship between a set of input data values and a set of result values, where the first input value (or subject argument) has the same type, or is a subtype of the subject type (also called the subject parameter), of the method. For example, a method called CITY, of type ADDRESS, can be passed input data values of type VARCHAR, and the result is an ADDRESS (or a subtype of ADDRESS).

Methods are defined implicitly or explicitly, as part of the definition of a user-defined structured type.

Implicitly defined methods are created for every structured type. Observer methods are defined for each attribute of the structured type. Observer methods allow applications to get the value of an attribute for an instance of the type. Mutator methods are also defined for each attribute, allowing applications to mutate the type instance by changing the value for an attribute of a type instance. The CITY method described above is an example of a mutator method for the type ADDRESS.

Explicitly defined methods, or user-defined methods, are methods that are registered to a database in SYSCAT.ROUTINES, by using a combination of CREATE TYPE (or ALTER TYPE ADD METHOD) and CREATE METHOD statements. All methods defined for a structured type are defined in the same schema as the type.

User-defined methods for structured types extend the function of the database system by adding method definitions (provided by users or third party vendors) that can be applied to structured type instances in the database engine. Defining database methods lets the database exploit the same methods in the engine that an application uses, providing more synergy between application and database.

External and SQL user-defined methods

A user-defined method can be either external or based on an SQL expression. An external method is defined to the database with a reference to an object code library and a function within that library that will be executed when the method is invoked. A method based on an SQL expression returns the result of the SQL expression when the method is invoked. Such methods do not require any object code library, because they are written completely in SQL.

A user-defined method can return a single-valued answer each time it is called. This value can be a structured type. A method can be defined as type preserving (using SELF AS RESULT), to allow the dynamic type of the subject argument to be returned as the returned type of the method. All implicitly defined mutator methods are type preserving.

Method signatures

A method is identified by its subject type, a method name, the number of parameters, and the data types of its parameters. This is called a method signature, and it must be unique within the database.

There can be more than one method with the same name for a structured type, provided that:
- The number of parameters or the data types of the parameters are different, or
Methods

- The methods are part of the same method hierarchy (that is, the methods are in an overriding relationship or override the same original method), or
- The same function signature (using the subject type or any of its subtypes or supertypes as the first parameter) does not exist.

A method name that has multiple method instances is called an overloaded method. A method name can be overloaded within a type, in which case there is more than one method by that name for the type (all of which have different parameter types). A method name can also be overloaded in the subject type hierarchy, in which case there is more than one method by that name in the type hierarchy. These methods must have different parameter types.

A method can be invoked by referring (in an allowable context) to the method name, preceded by both a reference to a structured type instance (the subject argument), and the double dot operator. A list of arguments enclosed in parentheses must follow. Which method is actually invoked depends on the static type of the subject type, using the method resolution process described in the following section. Methods defined WITH FUNCTION ACCESS can also be invoked using function invocation, in which case the regular rules for function resolution apply.

If function resolution results in a method defined WITH FUNCTION ACCESS, all subsequent steps of method invocation are processed.

Access to methods is controlled through the EXECUTE privilege. GRANT and REVOKE statements are used to specify who can or cannot execute a specific method or a set of methods. The EXECUTE privilege (or DATAACCESS authority) is needed to invoke a method. The definer of the method automatically receives the EXECUTE privilege. The definer of an external method or an SQL method having the WITH GRANT option on all underlying objects also receives the WITH GRANT option with the EXECUTE privilege on the method. The definer (or authorization ID with the ACCESSCTRL or SECADM authority) must then grant it to the user who wants to invoke the method from any SQL statement, or reference the method in any DDL statement (such as CREATE VIEW, CREATE TRIGGER, or when defining a constraint). If the EXECUTE privilege is not granted to a user, the method will not be considered by the method resolution algorithm, even if it is a better match.

Method resolution

After method invocation, the database manager must decide which of the possible methods with the same name is the “best fit”. Functions (built-in or user-defined) are not considered during method resolution.

An argument is a value passed to a method upon invocation. When a method is invoked in SQL, it is passed the subject argument (of some structured type) and a list of zero or more arguments. They are positional in that the semantics of an argument are determined by its position in the argument list. A parameter is a formal definition of an input to a method. When a method is defined to the database, either implicitly (system-generated for a type) or by a user (a user-defined method), its parameters are specified (with the subject parameter as the first parameter), and the order of their definitions defines their positions and their semantics. Therefore, every parameter is a particular positional input to a method. On invocation, an argument corresponds to a particular parameter by virtue of its position in the list of arguments.
Methods

The database manager uses the name of the method given in the invocation, EXECUTE privilege on the method, the number and data types of the arguments, all the methods with the same name for the subject argument’s static type (and its supertypes), and the data types of their corresponding parameters as the basis for deciding whether or not to select a method. The following are the possible outcomes of the decision process:

- A particular method is deemed to be the best fit. For example, given the methods named RISK for the type SITE with signatures defined as:

```
PROXIMITY(INTEGER) FOR SITE
PROXIMITY(DOUBLE) FOR SITE
```

the following method invocation (where ST is a SITE column, DB is a DOUBLE column):

```sql
SELECT ST..PROXIMITY(DB) ...
```

then, the second PROXIMITY will be chosen.

The following method invocation (where SI is a SMALLINT column):

```sql
SELECT ST..PROXIMITY(SI) ...
```

would choose the first PROXIMITY, because SMALLINT can be promoted to INTEGER and is a better match than DOUBLE, which is further down the precedence list.

When considering arguments that are structured types, the precedence list includes the supertypes of the static type of the argument. The best fit is the function defined with the supertype parameter that is closest in the structured type hierarchy to the static type of the function argument.

- No method is deemed to be an acceptable fit. For example, given the same two functions in the previous case and the following function reference (where C is a CHAR(5) column):

```sql
SELECT ST..PROXIMITY(C) ...
```

the argument is inconsistent with the parameter of both PROXIMITY functions.

- A particular method is selected based on the methods in the type hierarchy and the number and data types of the arguments passed on invocation. For example, given methods named RISK for the types SITE and DRILLSITE (a subtype of SITE) with signatures defined as:

```
RISK(INTEGER) FOR DRILLSITE
RISK(DOUBLE) FOR SITE
```

and the following method invocation (where DRST is a DRILLSITE column, DB is a DOUBLE column):

```sql
SELECT DRST..RISK(DB) ...
```

the second RISK will be chosen, because DRILLSITE can be promoted to SITE.

The following method reference (where SI is a SMALLINT column):

```sql
SELECT DRST..RISK(SI) ...
```

would choose the first RISK, because SMALLINT can be promoted to INTEGER, which is closer on the precedence list than DOUBLE, and DRILLSITE is a better match than SITE, which is a supertype.

Methods within the same type hierarchy cannot have the same signatures, considering parameters other than the subject parameter.
Methods

Determining the best fit

A comparison of the data types of the arguments with the defined data types of the parameters of the methods under consideration forms the basis for the decision of which method in a group of like-named methods is the “best fit”. Note that the data types of the results of the methods under consideration do not enter into this determination.

For method resolution, whether the data type of the input arguments can be promoted to the data type of the corresponding parameter is considered when determining the best fit. Unlike function resolution, whether the input arguments can be implicitly cast to the data type of the corresponding parameter is not considered when determining the best fit. Modules are not considered during method resolution because methods cannot be defined in modules.

Method resolution is performed using the following steps:

1. First, find all methods from the catalog (SYSCAT.ROUTINES) such that all of the following are true:
   - The method name matches the invocation name, and the subject parameter is the same type or is a supertype of the static type of the subject argument.
   - The invoker has the EXECUTE privilege on the method.
   - The number of defined parameters matches the invocation.
   - Each invocation argument matches the method’s corresponding defined parameter in data type, or is “promotable” to it.

2. Next, consider each argument of the method invocation, from left to right. The leftmost argument (and thus the first argument) is the implicit SELF parameter. For example, a method defined for type ADDRESS_T has an implicit first parameter of type ADDRESS_T. For each argument, eliminate all functions that are not the best match for that argument. The best match for a given argument is the first data type appearing in the precedence list corresponding to the argument data type for which there exists a function with a parameter of that data type. Length, precision, scale, and the FOR BIT DATA attribute are not considered in this comparison. For example, a DECIMAL(9,1) argument is considered an exact match for a DECIMAL(6,5) parameter, a DECFLOAT(34) argument is considered an exact match for a DECFLOAT(16) parameter, and a VARCHAR(19) argument is an exact match for a VARCHAR(6) parameter. The best match for a user-defined structured-type argument is itself; the next best match is its immediate supertype, and so on for each supertype of the argument. Note that only the static type (declared type) of the structured-type argument is considered, not the dynamic type (most specific type).

3. At most, one candidate method remains after Step 2. This is the method that is chosen.

4. If there are no candidate methods remaining after step 2, an error is returned (SQLSTATE 42884).

Example of method resolution

Following is an example of successful method resolution.

There are seven FOO methods for three structured types defined in a hierarchy of GOVERNOR as a subtype of EMPEROR as a subtype of HEADOFSTATE, registered with the following signatures:
The method reference is as follows (where I1 and I2 are INTEGER columns, D is a DECIMAL column and E is an EMPEROR column):

```
SELECT E..FOO(I1, I2, D) ...
```

Following through the algorithm...

- FOO_7 is eliminated as a candidate, because the type GOVERNOR is a subtype (not a supertype) of EMPEROR.
- FOO_3 is eliminated as a candidate, because it has the wrong number of parameters.
- FOO_1 and FOO_6 are eliminated because, in both cases, the first argument (not the subject argument) cannot be promoted to the data type of the first parameter. Because there is more than one candidate remaining, the arguments are considered in order.
- For the subject argument, FOO_2 is a supertype, while FOO_4 and FOO_5 match the subject argument.
- For the first argument, the remaining methods, FOO_4 and FOO_5, are an exact match with the argument type. No methods can be eliminated from consideration; therefore the next argument must be examined.
- For this second argument, FOO_5 is an exact match, but FOO_4 is not, so it is eliminated from consideration. This leaves FOO_5 as the method chosen.

**Method invocation**

Once the method is selected, there are still possible reasons why the use of the method may not be permitted.

Each method is defined to return a result with a specific data type. If this result data type is not compatible with the context in which the method is invoked, an error will occur. For example, assume that the following methods named STEP are defined, each with a different data type as the result:

```
STEP(SMALLINT) FOR TYPEA RETURNS CHAR(5)
STEP(DOUBLE) FOR TYPEA RETURNS INTEGER
```

and the following method reference (where S is a SMALLINT column and TA is a column of TYPEA):

```
SELECT 3 + TA..STEP(S) ...
```

then, because there is an exact match on argument type, the first STEP is chosen. An error occurs on the statement, because the result type is CHAR(5) instead of a numeric type, as required for an argument of the addition operator.

Starting from the method that has been chosen, the algorithm described in “Dynamic dispatch of methods” is used to build the set of dispatchable methods at compile time. Exactly which method is invoked is described in “Dynamic dispatch of methods”.

Note that when the selected method is a type preserving method:
Methods

- the static result type following function resolution is the same as the static type of the subject argument of the method invocation
- the dynamic result type when the method is invoked is the same as the dynamic type of the subject argument of the method invocation.

This may be a subtype of the result type specified in the type preserving method definition, which in turn may be a supertype of the dynamic type that is actually returned when the method is processed.

In cases where the arguments of the method invocation were not an exact match to the data types of the parameters of the selected method, the arguments are converted to the data type of the parameter at execution using the same rules as assignment to columns. This includes the case where precision, scale, or length differs between the argument and the parameter, but excludes the case where the dynamic type of the argument is a subtype of the parameter’s static type.

Dynamic dispatch of methods

Methods provide the functionality and encapsulate the data of a type. A method is defined for a type and can always be associated with this type. One of the method’s parameters is the implicit SELF parameter. The SELF parameter is of the type for which the method has been declared. The argument that is passed as the SELF argument when the method is invoked in a DML statement is called subject.

When a method is chosen using method resolution (see “Method resolution” on page 205), or a method has been specified in a DDL statement, this method is known as the “most specific applicable authorized method”. If the subject is of a structured type, that method could have one or more overriding methods. DB2 must then determine which of these methods to invoke, based on the dynamic type (most specific type) of the subject at run time. This determination is called “determining the most specific dispatchable method”. That process is described here.

1. Find the original method in the method hierarchy that the most specific applicable authorized method is part of. This is called the root method.
2. Create the set of dispatchable methods, which includes the following:
   - The most specific applicable authorized method.
   - Any method that overrides the most specific applicable authorized method, and which is defined for a type that is a subtype of the subject of this invocation.
3. Determine the most specific dispatchable method, as follows:
   a. Start with an arbitrary method that is an element of the set of dispatchable methods and that is a method of the dynamic type of the subject, or of one of its supertypes. This is the initial most specific dispatchable method.
   b. Iterate through the elements of the set of dispatchable methods. For each method: If the method is defined for one of the proper subtypes of the type for which the most specific dispatchable method is defined, and if it is defined for one of the supertypes of the most specific type of the subject, then repeat step 2 with this method as the most specific dispatchable method; otherwise, continue iterating.
4. Invoke the most specific dispatchable method.

Example:
Given are three types, "Person", "Employee", and "Manager". There is an original method "income", defined for "Person", which computes a person's income. A person is by default unemployed (a child, a retiree, and so on). Therefore, "income" for type "Person" always returns zero. For type "Employee" and for type "Manager", different algorithms have to be applied to calculate the income. Hence, the method "income" for type "Person" is overridden in "Employee" and "Manager".

Create and populate a table as follows:

```
CREATE TABLE aTable (id integer, personColumn Person);
INSERT INTO aTable VALUES (0, Person()), (1, Employee()), (2, Manager());
```

List all persons who have a minimum income of $40000:

```
SELECT id, person, name
FROM aTable
WHERE person..income() >= 40000;
```

The method "income" for type "Person" is chosen, using method resolution, to be the most specific applicable authorized method.

1. The root method is "income" for "Person" itself.
2. The second step of the algorithm above is carried out to construct the set of dispatchable methods:
   - The method "income" for type "Person" is included, because it is the most specific applicable authorized method.
   - The method "income" for type "Employee", and "income" for "Manager" is included, because both methods override the root method, and both "Employee" and "Manager" are subtypes of "Person".
   Therefore, the set of dispatchable methods is: {"income" for "Person", "income" for "Employee", "income" for "Manager"}.
3. Determine the most specific dispatchable method:
   - For a subject whose most specific type is "Person":
     a. Let the initial most specific dispatchable method be "income" for type "Person".
     b. Because there is no other method in the set of dispatchable methods that is defined for a proper subtype of "Person" and for a supertype of the most specific type of the subject, "income" for type "Person" is the most specific dispatchable method.
   - For a subject whose most specific type is "Employee":
     a. Let the initial most specific dispatchable method be "income" for type "Person".
     b. Iterate through the set of dispatchable methods. Because method "income" for type "Employee" is defined for a proper subtype of "Person" and for a supertype of the most specific type of the subject (Note: A type is its own super- and subtype.), method "income" for type "Employee" is a better match for the most specific dispatchable method. Repeat this step with method "income" for type "Employee" as the most specific dispatchable method.
     c. Because there is no other method in the set of dispatchable methods that is defined for a proper subtype of "Employee" and for a supertype of the most specific type of the subject, method "income" for type "Employee" is the most specific dispatchable method.
   - For a subject whose most specific type is "Manager":
Methods

a. Let the initial most specific dispatchable method be "income" for type "Person".

b. Iterate through the set of dispatchable methods. Because method "income" for type "Manager" is defined for a proper subtype of "Person" and for a supertype of the most specific type of the subject (Note: A type is its own super- and subtype.), method "income" for type "Manager" is a better match for the most specific dispatchable method. Repeat this step with method "income" for type "Manager" as the most specific dispatchable method.

c. Because there is no other method in the set of dispatchable methods that is defined for a proper subtype of "Manager" and for a supertype of the most specific type of the subject, method "income" for type "Manager" is the most specific dispatchable method.

4. Invoke the most specific dispatchable method.
Conservative binding semantics

Object resolution takes place when defining an SQL object or processing a package bind operation.

The database manager chooses which particular defined SQL object to use for an SQL object referenced in a DDL statement or coded in an application.

At a later time, the database manager might resolve to a different SQL object, even though the original SQL object did not change in any way. This resolution to a different SQL object happens as a result of defining another SQL object (or adding a privilege to an existing function) that the object resolution algorithm defines as resolved ahead of the SQL object originally chosen. Examples of SQL objects and situations to which this resolution to a different SQL object applies include the following:

- Routines - a new routine could be defined that is a better fit or that is an equally good fit but earlier in the SQL path; or a privilege could be granted to an existing routine that is a better fit or that is an equally good fit but earlier in the SQL path
- User-defined data types - a new user-defined data type could be defined with the same name and in a schema that is earlier in the SQL path
- Global variables - a new global variable could be defined with the same name and in a schema that is earlier in the SQL path
- Tables or views that resolve using a public alias - an actual table, view, or private alias could be defined with the same name in the current schema
- Sequences that resolve using a public sequence alias - an actual sequence or private sequence alias could be defined with the same name in the current schema
- Modules that resolve to a public module alias - an actual module or private module alias could be defined with the same name in a schema that is in the SQL path

There are instances where the database manager must be able to repeat the resolution of SQL objects as originally resolved when the statement was processed. This is true when the following static objects are used:

- Static DML statements in packages
- Views
- Triggers
- Check constraints
- SQL routines
- Global variables with a user-defined type or default expression
- Routines with a user-defined parameter type or default expression

For static DML statements in packages, SQL object references are resolved during a bind operation. SQL object references in views, triggers, SQL routines, and check constraints are resolved when the SQL object is defined or revalidated. When an existing static object is used, conservative binding semantics are applied unless the object has been marked invalid or inoperative by a change in the database schema.

Conservative binding semantics ensure that SQL object references will be resolved using the same SQL path, default schema, and set of routines that were used when it was previously resolved. It also ensures that the timestamp of the definition of the SQL objects considered during conservative binding resolution is not later than
the timestamp when the statement was last bound or validated using non-conservative binding semantics. Non-conservative binding semantics use the same SQL path and default schema as the original generation of the package or statement, but does not consider the timestamp of the definition of the SQL objects and does not consider any previously resolved set of routines.

Some changes to the database schema, such as dropping objects, altering objects, or revoking privileges, can impact an SQL object so that the database manager can no longer resolve all dependent SQL objects of an existing SQL object using conservative binding semantics.

- When this happens for a static statement in an SQL package, the package is marked inoperative. The next use of a statement in this package will cause an implicit rebind of the package using non-conservative binding semantics in order to be able to resolve to SQL objects considering the latest changes in the database schema that caused that package to become inoperative.

- When this happens for a view, trigger, check constraint, or SQL routine, the SQL object is marked invalid. The next use of the object causes an implicit revalidation of the SQL object using non-conservative binding semantics.

Consider a database with two functions that have the signatures SCHEMA1.BAR(INTEGER) and SCHEMA2.BAR(DOUBLE). Assume the SQL path contains both schemas SCHEMA1 and SCHEMA2 (although their order within the SQL path does not matter). USER1 has been granted the EXECUTE privilege on the function SCHEMA2.BAR(DOUBLE). Suppose USER1 creates a view that invokes BAR(INT_VAL), where INT_VAL is a column or global variable defined with the INTEGER data type. This function reference in the view resolves to the function SCHEMA2.BAR(DOUBLE) because USER1 does not have the EXECUTE privilege on SCHEMA1.BAR(INTEGER). If USER1 is granted the EXECUTE privilege on SCHEMA1.BAR(INTEGER) after the view has been created, the view will continue to use SCHEMA2.BAR(DOUBLE) unless a database schema change causes the view to be marked invalid. The view is marked invalid if a required privilege is revoked or an object it depends on gets dropped or altered.

For static DML in packages, packages can rebind implicitly, or by explicitly issuing the REBIND command (or corresponding API), or the BIND command (or corresponding API). The implicit rebind is performed with conservative binding semantics if the package is marked invalid, but uses non-conservative binding semantics when the package is marked inoperative. A package is marked invalid only if an index on which it depends is dropped or altered. The REBIND command provides the option to resolve with conservative binding semantics (RESOLVE CONSERVATIVE) or to resolve by considering new routines, data types, or global variables (RESOLVE ANY, which is the default option). The RESOLVE CONSERVATIVE option can be used only if the package has not been marked inoperative by the database manager (SQLSTATE 51028).

The description of conservative binding semantics in this topic has assumed that the database configuration parameter auto_reval has a setting other than DISABLED; the default for new databases is DEFERRED; the default for databases upgraded to Version 9.7 is DISABLED. If auto_reval is set to DISABLED, then conservative binding semantics, invalidation, and revalidation behavior are the same as in releases previous to Version 9.7. Under this setting, conservative binding semantics only considers the timestamp of the definition of the SQL objects for functions, methods, user-defined types, and global variables. For invalidation and revalidation behavior, this means, in the case of the DROP, REVOKE, and ALTER statements, that either the semantics are more restrictive or
Conservative binding semantics

the impact on dependent objects is to cascade and drop the object. In the case of packages, most database schema changes result in marking the package invalid and using conservative binding semantics during implicit rebind. However, when the schema is changed by dropping a dependent function and auto_reval is set to DISABLED, the package dependent on the function is marked inoperative and there is no implicit rebind of the inoperative package.
Expressions

An expression specifies a value. It can be a simple value, consisting of only a constant or a column name, or it can be more complex. When repeatedly using similar complex expressions, an SQL function to encapsulate a common expression can be considered.

Authorization

The use of some of the expressions, such as a scalar-subselect, sequence-reference, or function-invocation may require having the appropriate authorization. For these expressions, the privileges held by the authorization ID of the statement must include the following authorization:

- scalar-subselect. For information about authorization considerations, see: Queries
- sequence-reference. The authorization to reference the sequence. For information about authorization considerations, see: Sequence authorization.
- function-invocation. The authorization to execute a user-defined function. For information about authorization considerations, see Function invocation.
- variable. If the variable is a global variable, the authorization to reference the global variable is required. For information, see Global variables.

In a Unicode database, an expression that accepts a character or graphic string will accept any string types for which conversion is supported.

expression:
Expressions

operator:

- operator: (19)
  - CONCAT
  - / (20)
  - * (20)
  - + (20)
  - - (20)

function invocation:

- (expression) (21)
- constant (22)
- column name (23)
- variable (24)
- special register (25)
- scalar full select (26)
- labeled duration (27)
- case expression (28)
- cast specification (29)
- field reference (30)
- xml cast specification (31)
- array element specification (32)
- array constructor (33)
- dereference operation (34)
- method invocation (35)
- OLAP specification (36)
- row change expression (37)
- sequence reference (38)
- subtype treatment (39)

Notes:

1. See “Function invocation” on page 191 for more information.
2. See “Constants” on page 144 for more information.
3. See “References to variables” on page 70 for more information.
4. See “Special registers” on page 149 for more information.
5. See “Scalar full select” on page 225 for more information.
Expressions without operators

If no operators are used, the result of the expression is the specified value.

Examples:

\[
\text{SALARY:SALARY'SALARY'}
\]

\[
\text{MAX(SALARY)}
\]

Expressions with the concatenation operator

The concatenation operator (CONCAT) combines two operands to form a \textit{string expression}.

The first operand is an expression that returns a value of any string data type, any numeric data type, or any datetime data type. The second operand is also an expression that returns a value of any string data type, any numeric data type, or any datetime data type. However, some data types are not supported in combination with the data type of the first operand, as described below.

The operands can be any combination of string (except binary string), numeric, and datetime values. When any operand is a non-string value, it is implicitly cast to VARCHAR. A binary string can only be concatenated with another binary string. However, through the castable process of function resolution, a binary string can be concatenated with a character string defined as FOR BIT DATA when the first operand is the binary string.

Concatenation involving both character string operands and graphic string operands is supported only in a Unicode database. Character operands are first converted to the graphic data type before the concatenation. Character strings defined as FOR BIT DATA cannot be cast to the graphic data type.

If either operand can be null, the result can be null, and if either is null, the result is the null value. Otherwise, the result consists of the first operand string followed by the second. Note that no check is made for improperly formed mixed data when doing concatenation.
Expressions

The length of the result is the sum of the lengths of the operands.

The data type and length attribute of the result is determined from that of the operands as shown in the following table:

Table 22. Data Type and Length of Concatenated Operands

<table>
<thead>
<tr>
<th>Operands</th>
<th>Combined Length Attributes</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR(A) CHAR(B)</td>
<td>&lt;255</td>
<td>CHAR(A+B)</td>
</tr>
<tr>
<td>CHAR(A) CHAR(B)</td>
<td>&gt;254</td>
<td>VARCHAR(A+B)</td>
</tr>
<tr>
<td>CHAR(A) VARCHAR(B)</td>
<td>&lt;4001</td>
<td>VARCHAR(A+B)</td>
</tr>
<tr>
<td>CHAR(A) VARCHAR(B)</td>
<td>&gt;4000</td>
<td>LONG VARCHAR</td>
</tr>
<tr>
<td>CHAR(A) LONG VARCHAR</td>
<td></td>
<td>LONG VARCHAR</td>
</tr>
<tr>
<td>VARCHAR(A) VARCHAR(B)</td>
<td>&lt;4001</td>
<td>VARCHAR(A+B)</td>
</tr>
<tr>
<td>VARCHAR(A) VARCHAR(B)</td>
<td>&gt;4000</td>
<td>LONG VARCHAR</td>
</tr>
<tr>
<td>VARCHAR(A) LONG VARCHAR</td>
<td></td>
<td>LONG VARCHAR</td>
</tr>
<tr>
<td>LONG VARCHAR LONG VARCHAR</td>
<td></td>
<td>LONG VARCHAR</td>
</tr>
<tr>
<td>CLOB(A) CHAR(B)</td>
<td></td>
<td>CLOB(MIN(A+B, 2G))</td>
</tr>
<tr>
<td>CLOB(A) VARCHAR(B)</td>
<td></td>
<td>CLOB(MIN(A+B, 2G))</td>
</tr>
<tr>
<td>CLOB(A) LONG VARCHAR</td>
<td></td>
<td>CLOB(MIN(A+32K, 2G))</td>
</tr>
<tr>
<td>CLOB(A) CLOB(B)</td>
<td></td>
<td>CLOB(MIN(A+B, 2G))</td>
</tr>
<tr>
<td>GRAPHIC(A) GRAPHIC(B)</td>
<td>&lt;128</td>
<td>GRAPHIC(A+B)</td>
</tr>
<tr>
<td>GRAPHIC(A) GRAPHIC(B)</td>
<td>&gt;127</td>
<td>VARGRAPHIC(A+B)</td>
</tr>
<tr>
<td>GRAPHIC(A) VARGRAPHIC(B)</td>
<td>&lt;2001</td>
<td>VARGRAPHIC(A+B)</td>
</tr>
<tr>
<td>GRAPHIC(A) VARGRAPHIC(B)</td>
<td>&gt;2000</td>
<td>LONG VARGRAPHIC</td>
</tr>
<tr>
<td>GRAPHIC(A) LONG VARGRAPHIC</td>
<td></td>
<td>LONG VARGRAPHIC</td>
</tr>
<tr>
<td>VARGRAPHIC(A) VARGRAPHIC(B)</td>
<td>&lt;2001</td>
<td>VARGRAPHIC(A+B)</td>
</tr>
<tr>
<td>VARGRAPHIC(A) VARGRAPHIC(B)</td>
<td>&gt;2000</td>
<td>LONG VARGRAPHIC</td>
</tr>
<tr>
<td>VARGRAPHIC(A) LONG VARGRAPHIC</td>
<td></td>
<td>LONG VARGRAPHIC</td>
</tr>
<tr>
<td>LONG VARGRAPHIC LONG VARGRAPHIC</td>
<td></td>
<td>LONG VARGRAPHIC</td>
</tr>
<tr>
<td>DBCLOB(A) GRAPHIC(B)</td>
<td></td>
<td>DBCLOB(MIN(A+B, 1G))</td>
</tr>
<tr>
<td>DBCLOB(A) VARGRAPHIC(B)</td>
<td></td>
<td>DBCLOB(MIN(A+B, 1G))</td>
</tr>
<tr>
<td>DBCLOB(A) LONG VARGRAPHIC</td>
<td></td>
<td>DBCLOB(MIN(A+16K, 1G))</td>
</tr>
<tr>
<td>DBCLOB(A) DBCLOB(B)</td>
<td></td>
<td>DBCLOB(MIN(A+B, 1G))</td>
</tr>
<tr>
<td>BLOB(A) BLOB(B)</td>
<td></td>
<td>BLOB(MIN(A+B, 2G))</td>
</tr>
</tbody>
</table>

Note that, for compatibility with previous versions, there is no automatic escalation of results involving LONG VARCHAR or LONG VARGRAPHIC data types to LOB data types. For example, concatenation of a CHAR(200) value and a completely full LONG VARCHAR value would result in an error rather than in a promotion to a CLOB data type.

The code page of the result is considered a derived code page and is determined by the code page of its operands.
Expressions

One operand may be a parameter marker. If a parameter marker is used, then the data type and length attributes of that operand are considered to be the same as those for the non-parameter marker operand. The order of operations must be considered to determine these attributes in cases with nested concatenation.

Example 1: If FIRSTNME is Pierre and LASTNAME is Fermat, then the following:

```
FIRSTNME CONCAT ' ' CONCAT LASTNAME
```

returns the value Pierre Fermat.

Example 2: Given:
- COLA defined as VARCHAR(5) with value 'AA'
- :host_var defined as a character host variable with length 5 and value 'BB'
- COLC defined as CHAR(5) with value 'CC'
- COLD defined as CHAR(5) with value 'DDDD'

The value of COLA CONCAT :host_var CONCAT COLC CONCAT COLD is

'AABB CC DDDDD'

The data type is VARCHAR, the length attribute is 17 and the result code page is the section code page. For more information on section code pages, see "Derivation of code page values".

Example 3: Given:
- COLA defined as CHAR(10)
- COLB defined as VARCHAR(5)

The parameter marker in the expression:

```
COLA CONCAT COLB CONCAT ?
```

is considered VARCHAR(15), because COLA CONCAT COLB is evaluated first, giving a result that is the first operand of the second CONCAT operation.

User-defined types

A user-defined type cannot be used with the concatenation operator, even if it is a distinct type with a source data type that is a string type. To concatenate, create a function with the CONCAT operator as its source. For example, if there were distinct types TITLE and TITLE_DESCRIPTION, both of which had VARCHAR(25) data types, the following user-defined function, ATTACH, could be used to concatenate them.

```
CREATE FUNCTION ATTACH (TITLE, TITLE_DESCRIPTION)
    RETURNS VARCHAR(50) SOURCE CONCAT (VARCHAR(), VARCHAR())
```

Alternately, the concatenation operator could be overloaded using a user-defined function to add the new data types.

```
CREATE FUNCTION CONCAT (TITLE, TITLE_DESCRIPTION)
    RETURNS VARCHAR(50) SOURCE CONCAT (VARCHAR(), VARCHAR())
```

Expressions with arithmetic operators

If arithmetic operators are used, the result of the expression is a value derived from the application of the operators to the values of the operands.
Expressions

If any operand can be null, or the database is configured with `dft_sqlmathwarn` set to yes, the result can be null.

If any operand has the null value, the result of the expression is the null value.

Arithmetic operators can be applied to signed numeric types and datetime types (see [“Datetime arithmetic in SQL” on page 227](#)). For example, USER+2 is invalid. Sourced functions can be defined for arithmetic operations on distinct types with a source type that is a signed numeric type.

The prefix operator + (unary plus) does not change its operand. The prefix operator - (unary minus) reverses the sign of a nonzero non decimal floating-point operand. The prefix operator - (unary minus) reverses the sign of all decimal floating-point operands, including zero and special values; that is, signalling and non-signalling NaNs and plus and minus infinity. If the data type of A is small integer, the data type of -A is large integer. The first character of the token following a prefix operator must not be a plus or minus sign.

The *infix operators* +, -, *, and / specify addition, subtraction, multiplication, and division, respectively. The value of the second operand of division must not be zero, except if the calculation is performed using decimal floating-point arithmetic. These operators can also be treated as functions. Thus, the expression a+b is equivalent to the expression “operator” function.

Operands with a character or graphic string data type, except LOBs, are converted to DECIMAL(34) using the rules for CAST specification, prior to performing the arithmetic operation. For more information, see “Casting between data types”. Note that arithmetic involving graphic string operands is supported only in a Unicode database.

Operands with a string data type are converted to DECIMAL(34) using the rules for CAST specification prior to performing the arithmetic operation. For more information, refer to “Casting between data types”. The string must contain a valid representation of a number.

**Arithmetic errors**

If an arithmetic error such as divide by zero or a numeric overflow occurs during the processing of an non-decimal floating-point expression, an error is returned (SQLSTATE 22003 or 22012). For decimal floating-point expressions, a warning is returned (SQLSTATEs 0168C, 0168D, 0168E, or 0168F) which depends on the nature of the arithmetic condition.

A database can be configured (using `dft_sqlmathwarn` set to yes) so that arithmetic errors return a null value for the non-decimal floating-point expression, the query returns a warning (SQLSTATE 01519 or 01564), and proceeds with processing the SQL statement.

For decimal floating-point expressions, `dft_sqlmathwarn` has no effect; arithmetic conditions return an appropriate value (possibly a decimal floating-point special value), the query returns a warning (SQLSTATEs 0168C, 0168D, 0168E, or 0168F), and proceeds with processing of the SQL statement. Special values returned include plus and minus infinity and not a number. Arithmetic expressions involving one or more decimal floating-point numbers never evaluate to a null value unless one or more of the arguments to the expression are null.
Expressions

When arithmetic errors are treated as nulls, there are implications on the results of SQL statements. The following are some examples of these implications.

- An arithmetic error that occurs in the expression that is the argument of an aggregate function causes the row to be ignored in the determining the result of the aggregate function. If the arithmetic error was an overflow, this may significantly impact the result values.
- An arithmetic error that occurs in the expression of a predicate in a WHERE clause can cause rows not to be included in the result.
- An arithmetic error that occurs in the expression of a predicate in a check constraint results in the update or insert proceeding since the constraint is not false.

If these types of impacts are not acceptable, additional steps should be taken to handle the arithmetic error to produce acceptable results. Some examples are:

- add a case expression to check for zero divide and set the desired value for such a situation
- add additional predicates to handle nulls (like a check constraint on not nullable columns could become:
  \[
  \text{check (c1*c2 is not null and c1*c2>5000)}
  \]
  to cause the constraint to be violated on an overflow).

**Two integer operands**

If both operands of an arithmetic operator are integers, the operation is performed in binary and the result is a large integer unless either (or both) operand is a big integer, in which case the result is a big integer. Any remainder of division is lost. The result of an integer arithmetic operation (including unary minus) must be within the range of the result type.

**Integer and decimal operands**

If one operand is an integer and the other is a decimal, the operation is performed in decimal using a temporary copy of the integer that has been converted to a decimal number with precision \( p \) and scale 0; \( p \) is 19 for a big integer, 11 for a large integer, and 5 for a small integer.

**Two decimal operands**

If both operands are decimal, the operation is performed in decimal. The result of any decimal arithmetic operation is a decimal number with a precision and scale that are dependent on the operation and the precision and scale of the operands. If the operation is addition or subtraction and the operands do not have the same scale, the operation is performed with a temporary copy of one of the operands. The copy of the shorter operand is extended with trailing zeros so that its fractional part has the same number of digits as the longer operand.

The result of a decimal operation must not have a precision greater than 31. The result of decimal addition, subtraction, and multiplication is derived from a temporary result which may have a precision greater than 31. If the precision of the temporary result is not greater than 31, the final result is the same as the temporary result.
Decimal arithmetic in SQL

The following formulas define the precision and scale of the result of decimal operations in SQL. The symbols $p$ and $s$ denote the precision and scale of the first operand, and the symbols $p'$ and $s'$ denote the precision and scale of the second operand.

**Addition and subtraction**

The precision is $\min(31, \max(p-s, p'-s') + \max(s, s')+1)$. The scale of the result of addition and subtraction is $\max(s, s')$.

**Multiplication**

The precision of the result of multiplication is $\min(31, p + p')$ and the scale is $\min(31, s + s')$.

**Division**

The precision of the result of division is 31. The scale is $31-p+s-s'$. The scale must not be negative.

**Note:** The `min_dec_div_3` database configuration parameter alters the scale for decimal arithmetic operations involving division. If the parameter value is set to NO, the scale is calculated as $31-p+s-s'$. If the parameter is set to YES, the scale is calculated as $\max(3, 31-p+s-s')$. This ensures that the result of decimal division always has a scale of at least 3 (precision is always 31).

**Floating-point operands**

If either operand of an arithmetic operator is floating-point, but not decimal floating-point, the operation is performed in floating-point. The operands are first converted to double-precision floating-point numbers, if necessary. Thus, if any element of an expression is a floating-point number, the result of the expression is a double-precision floating-point number.

An operation involving a floating-point number and an integer is performed with a temporary copy of the integer which has been converted to double-precision floating-point. An operation involving a floating-point number and a decimal number is performed with a temporary copy of the decimal number which has been converted to double-precision floating-point. The result of a floating-point operation must be within the range of floating-point numbers.

The order in which floating-point operands (or arguments to functions) are processed can slightly affect results because floating-point operands are approximate representations of real numbers. Since the order in which operands are processed may be implicitly modified by the optimizer (for example, the optimizer may decide what degree of parallelism to use and what access plan to use), an application that uses floating-point operands should not depend on the results being precisely the same each time an SQL statement is executed.

**Decimal floating-point operands**

If either operand of an arithmetic operator is decimal floating-point, the operation is performed in decimal floating-point.
Expressions

Integer and decimal floating-point operands
If one operand is a small integer or large integer and the other is a DECFLOAT(\(n\)) number, the operation is performed in DECFLOAT(\(n\)) using a temporary copy of the integer that has been converted to a DECFLOAT(\(n\)) number. If one operand is a big integer, and the other is a decimal floating-point number, a temporary copy of the big integer is converted to a DECFLOAT(34) number. The rules for two-decimal floating-point operands then apply.

Decimal and decimal floating-point operands
If one operand is a decimal and the other is a decimal floating-point number, the operation is performed in decimal floating-point using a temporary copy of the decimal number that has been converted to a decimal floating-point number based on the precision of the decimal number. If the decimal number has a precision less than 17, the decimal number is converted to a DECFLOAT(16) number; otherwise, the decimal number is converted to a DECFLOAT(34) number. The rules for two-decimal floating-point operands then apply.

Floating-point and decimal floating-point operands
If one operand is a floating-point number (REAL or DOUBLE) and the other is a DECFLOAT(\(n\)) number, the operation is performed in decimal floating-point using a temporary copy of the floating-point number that has been converted to a DECFLOAT(\(n\)) number.

Two decimal floating-point operands
If both operands are DECFLOAT(\(n\)), the operation is performed in DECFLOAT(\(n\)). If one operand is DECFLOAT(16) and the other is DECFLOAT(34), the operation is performed in DECFLOAT(34).

General arithmetic operation rules for decimal floating-point
The following general rules apply to all arithmetic operations on the decimal floating-point data type:

- Every operation on finite numbers is carried out as though an exact mathematical result is computed, using integer arithmetic on the coefficient, where possible.

  If the coefficient of the theoretical exact result has no more than the number of digits that reflect its precision (16 or 34), it is used for the result without change (unless there is an underflow or overflow condition). If the coefficient has more than the number of digits that reflect its precision, it is rounded to exactly the number of digits that reflect its precision (16 or 34), and the exponent is increased by the number of digits that are removed.

  The CURRENT DECFLOAT Rounding Mode special register determines the rounding mode.

  If the value of the adjusted exponent of the result is less than \(E_{\text{min}}\), the calculated coefficient and exponent form the result, unless the value of the exponent is less than \(E_{\text{tiny}}\) in which case the exponent is set to \(E_{\text{tiny}}\), the coefficient is rounded (possibly to zero) to match the adjustment of the exponent, and the sign remains unchanged. If this rounding gives an inexact result, an underflow exception condition is returned.

  If the value of the adjusted exponent of the result is larger than \(E_{\text{max}}\) an overflow exception condition is returned. In this case, the result is defined as an overflow exception condition and might be infinite. It has the same sign as the theoretical result.
Arithmetic that uses the special value infinity follows the usual rules, where negative infinity is less than every finite number and positive infinity is greater than every finite number. Under these rules, an infinite result is always exact. Certain uses of infinity return an invalid operation condition. The following list shows the operations that can cause an invalid operation condition. The result of such an operation is NaN when one of the operands is infinity but the other operand is not NaN or sNaN.

- Add +infinity to -infinity during an addition or subtraction operation
- Multiply 0 by +infinity or -infinity
- Divide either +infinity or -infinity by either +infinity or -infinity
- Either argument of the QUANTIZE function is +infinity or -infinity
- The second argument of the POWER function is +infinity or -infinity
- Signaling NaNs used as operands to arithmetic operations

The following rules apply to arithmetic operations and the NaN value:

- The result of any arithmetic operation that has a NaN (quiet or signalling) operand is NaN. The sign of the result is copied from the first operand that is a signalling NaN; if neither operand is signalling, the sign is copied from the first operand that is a NaN. Whenever a result is a NaN, the sign of the result depends only on the copied operand.
- The sign of the result of a multiplication or division operation is negative only if the operands have different signs and neither is a NaN.
- The sign of the result of an addition or subtraction operation is negative only if the result is less than zero and neither operand is a NaN, except for the following cases where the result is a negative 0:
  - A result is rounded to zero, and the value, before rounding, had a negative sign
  - -0 is added to 0
  - 0 is subtracted from -0
- Operands with opposite signs are added, or operands with the same sign are subtracted; the result has a coefficient of 0, and the rounding mode is ROUND_FLOOR
- Operands are multiplied or divided, the result has a coefficient of 0, and the signs of the operands are different
- The first argument of the POWER function is -0, and the second argument is a positive odd number
- The argument of the CEIL, FLOOR, or SQRT function is -0
- The first argument of the ROUND or TRUNCATE function is -0

The following examples show special decimal floating-point values as operands:

```
INFINITY + 1 = INFINITY
INFINITY + INFINITY = INFINITY
INFINITY + -INFINITY = NaN -- warning
NAN + 1 = NAN
NAN + INFINITY = NAN
1 - INFINITY = -INFINITY
INFINITY - INFINITY = NaN -- warning
-INFINITY - -INFINITY = NAN -- warning
-0.0 - 0.0E1 = -0.0
-1.0 * 0.0E1 = -0.0
1.0E1 / 0 = INFINITY -- warning
-1.0E5 / 0.0 = -INFINITY -- warning
1.0E5 / -0 = -INFINITY -- warning
```
Expressions

\[ \begin{align*}
\text{INFINITY} / -\text{INFINITY} & = \text{NAN} \quad \text{-- warning} \\
\text{INFINITY} / 0 & = \text{INFINITY} \\
-\text{INFINITY} / 0 & = -\text{INFINITY} \\
-\text{INFINITY} / -0 & = \text{INFINITY}
\end{align*} \]

User-defined types as operands

A user-defined type cannot be used with arithmetic operators, even if its source data type is numeric. To perform an arithmetic operation, create a function with the arithmetic operator as its source. For example, if there were distinct types INCOME and EXPENSES, both of which had DECIMAL(8,2) data types, then the following user-defined function, REVENUE, could be used to subtract one from the other.

```
CREATE FUNCTION REVENUE (INCOME, EXPENSES)
    RETURNS DECIMAL(8,2)
    SOURCE "-" (DECIMAL, DECIMAL)
```

Alternately, the - (minus) operator could be overloaded using a user-defined function to subtract the new data types.

```
CREATE FUNCTION "-" (INCOME, EXPENSES)
    RETURNS DECIMAL(8,2)
    SOURCE "-" (DECIMAL, DECIMAL)
```

Precedence of operations

Expressions within parentheses and dereference operations are evaluated first from left to right. (Parentheses are also used in fullselects, search conditions, and functions. However, they should not be used to arbitrarily group sections within SQL statements.) When the order of evaluation is not specified by parentheses, prefix operators are applied before multiplication and division, and multiplication and division are applied before addition and subtraction. Operators at the same precedence level are applied from left to right.

```
1.10 * (Salary + Bonus) + Salary / :VAR3
```

Figure 11. Precedence of Operations

Scalar fullselect

Scalar fullselect:

```
---(---fullselect---)
```

A scalar fullselect is a fullselect, enclosed in parentheses, that returns a single row consisting of a single column value. If the fullselect does not return a row, the result of the expression is the null value. If the select list element is an expression that is simply a column name or a dereference operation, the result column name is based on the name of the column. The authorization required for a scalar fullselect is the same as that required for an SQL query.
Datetime operations and durations

Datetime values can be incremented, decremented, and subtracted. These operations can involve decimal numbers called durations. The following sections describe duration types and detail the rules for datetime arithmetic.

Durations

A duration is a number representing an interval of time. There are four types of durations.

labeled-duration:

- function
- (expression)
- constant
- column-name
- global-variable
- host-variable

A labeled duration represents a specific unit of time as expressed by a number (which can be the result of an expression) followed by one of the seven duration keywords: YEARS, MONTHS, DAYS, HOURS, MINUTES, SECONDS, or MICROSECONDS. (The singular form of these keywords is also acceptable: YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, and MICROSECOND.) The number specified is converted as if it were assigned to a DECIMAL(15,0) number, except for SECONDS which uses DECIMAL(27,12) to allow 0 to 12 digits of fractional seconds to be included. A labeled duration can only be used as an operand of an arithmetic operator in which the other operand is a value of data type DATE, TIME, or TIMESTAMP. Thus, the expression HIREDATE + 2 MONTHS + 14 DAYS is valid, whereas the expression HIREDATE + (2 MONTHS + 14 DAYS) is not. In both of these expressions, the labeled durations are 2 MONTHS and 14 DAYS.

A date duration represents a number of years, months, and days, expressed as a DECIMAL(8,0) number. To be properly interpreted, the number must have the format yyyyymmdd, where yyyy represents the number of years, mm the number of months, and dd the number of days. (The period in the format indicates a DECIMAL data type.) The result of subtracting one date value from another, as in the expression HIREDATE - BRTHDATE, is a date duration.

A time duration represents a number of hours, minutes, and seconds, expressed as a DECIMAL(6,0) number. To be properly interpreted, the number must have the format hhmmss, where hh represents the number of hours, mm the number of minutes, and ss the number of seconds. (The period in the format indicates a DECIMAL data type.) The result of subtracting one time value from another is a time duration.
Datet ime operations and durations

A timestamp duration represents a number of years, months, days, hours, minutes, seconds, and fractional seconds, expressed as a DECIMAL(14+s,s) number, where s is the number of digits of fractional seconds ranging from 0 to 12. To be properly interpreted, the number must have the format yyyyymmddhhmmss.nnnnnnnnnnnn, where yyyy, mm, dd, hh, mm, ss, and nnnnnnnnnnnn represent, respectively, the number of years, months, days, hours, minutes, seconds, and fractional seconds. The result of subtracting one timestamp value from another is a timestamp duration, with scale that matches the maximum timestamp precision of the timestamp operands.

Datet ime arithmetic in SQL

The only arithmetic operations that can be performed on datetime values are addition and subtraction. If a datetime value is the operand of addition, the other operand must be a duration. The specific rules governing the use of the addition operator with datetime values follow.

• If one operand is a date, the other operand must be a date duration or labeled duration of YEARS, MONTHS, or DAYS.
• If one operand is a time, the other operand must be a time duration or a labeled duration of HOURS, MINUTES, or SECONDS.
• If one operand is a timestamp, the other operand must be a duration. Any type of duration is valid.
• Neither operand of the addition operator can be a parameter marker.

The rules for the use of the subtraction operator on datetime values are not the same as those for addition because a datetime value cannot be subtracted from a duration, and because the operation of subtracting two datetime values is not the same as the operation of subtracting a duration from a datetime value. The specific rules governing the use of the subtraction operator with datetime values follow.

• If the first operand is a timestamp, the second operand must be a date, a timestamp, a string representation of a date, a string representation of a timestamp, or a duration.
• If the second operand is a timestamp, the first operand must be a date, a timestamp, a string representation of a date, or a string representation of a timestamp.
• If the first operand is a date, the second operand must be a date, a date duration, a string representation of a date, or a labeled duration of YEARS, MONTHS, or DAYS.
• If the second operand is a date, the first operand must be a date, or a string representation of a date.
• If the first operand is a time, the second operand must be a time, a time duration, a string representation of a time, or a labeled duration of HOURS, MINUTES, or SECONDS.
• If the second operand is a time, the first operand must be a time, or string representation of a time.
• Neither operand of the subtraction operator can be a parameter marker.

Date arithmetic

Dates can be subtracted, incremented, or decremented.

• The result of subtracting one date (DATE2) from another (DATE1) is a date duration that specifies the number of years, months, and days between the two dates. The data type of the result is DECIMAL(8,0). If DATE1 is greater than or
equal to DATE2, DATE2 is subtracted from DATE1. If DATE1 is less than DATE2, however, DATE1 is subtracted from DATE2, and the sign of the result is made negative. The following procedural description clarifies the steps involved in the operation result = DATE1 - DATE2.

If DAY(DATE2) <= DAY(DATE1)
then DAY(RESULT) = DAY(DATE1) - DAY(DATE2).

If DAY(DATE2) > DAY(DATE1)
then DAY(RESULT) = N + DAY(DATE1) - DAY(DATE2)
where N = the last day of MONTH(DATE2).
MONTH(DATE2) is then incremented by 1.

If MONTH(DATE2) <= MONTH(DATE1)
then MONTH(RESULT) = MONTH(DATE1) - MONTH(DATE2).

If MONTH(DATE2) > MONTH(DATE1)
then MONTH(RESULT) = 12 + MONTH(DATE1) - MONTH(DATE2).
YEAR(DATE2) is then incremented by 1.
YEAR(RESULT) = YEAR(DATE1) - YEAR(DATE2).

For example, the result of DATE('3/15/2000') - '12/31/1999' is 00000215. (or, a duration of 0 years, 2 months, and 15 days).

- The result of adding a duration to a date, or of subtracting a duration from a date, is itself a date. (For the purposes of this operation, a month denotes the equivalent of a calendar page. Adding months to a date, then, is like turning the pages of a calendar, starting with the page on which the date appears.) The result must fall between the dates January 1, 0001 and December 31, 9999 inclusive.

If a duration of years is added or subtracted, only the year portion of the date is affected. The month is unchanged, as is the day unless the result would be February 29 of a non-leap-year. In this case, the day is changed to 28, and a warning indicator in the SQLCA is set to indicate the adjustment.

Similarly, if a duration of months is added or subtracted, only months and, if necessary, years are affected. The day portion of the date is unchanged unless the result would be invalid (September 31, for example). In this case, the day is set to the last day of the month, and a warning indicator in the SQLCA is set to indicate the adjustment.

Adding or subtracting a duration of days will, of course, affect the day portion of the date, and potentially the month and year.

Date durations, whether positive or negative, may also be added to and subtracted from dates. As with labeled durations, the result is a valid date, and a warning indicator is set in the SQLCA whenever an end-of-month adjustment is necessary.

When a positive date duration is added to a date, or a negative date duration is subtracted from a date, the date is incremented by the specified number of years, months, and days, in that order. Thus, DATE1 + X, where X is a positive DECIMAL(8,0) number, is equivalent to the expression:

\[ \text{DATE1} + \text{YEAR(X)} \text{ YEARS} + \text{MONTH(X)} \text{ MONTHS} + \text{DAY(X)} \text{ DAYS} \]

When a positive date duration is subtracted from a date, or a negative date duration is added to a date, the date is decremented by the specified number of days, months, and years, in that order. Thus, DATE1 - X, where X is a positive DECIMAL(8,0) number, is equivalent to the expression:

\[ \text{DATE1} - \text{DAY(X)} \text{ DAYS} - \text{MONTH(X)} \text{ MONTHS} - \text{YEAR(X)} \text{ YEARS} \]

When adding durations to dates, adding one month to a given date gives the same date one month later unless that date does not exist in the later month. In that case, the date is set to that of the last day of the later month. For example, January 28 plus one month gives February 28; and one month added to January 29, 30, or 31 results in either February 28 or, for a leap year, February 29.
Datetime operations and durations

Note: If one or more months is added to a given date and then the same number of months is subtracted from the result, the final date is not necessarily the same as the original date.

Time arithmetic

Times can be subtracted, incremented, or decremented.

- The result of subtracting one time (TIME2) from another (TIME1) is a time duration that specifies the number of hours, minutes, and seconds between the two times. The data type of the result is DECIMAL(6,0).

  If TIME1 is greater than or equal to TIME2, TIME2 is subtracted from TIME1.

  If TIME1 is less than TIME2, however, TIME1 is subtracted from TIME2, and the sign of the result is made negative. The following procedural description clarifies the steps involved in the operation result = TIME1 - TIME2.

    IF SECOND(TIME2) <= SECOND(TIME1)
    then SECONDRESULT = SECOND(TIME1) - SECOND(TIME2).

    IF SECOND(TIME2) > SECOND(TIME1)
    then SECONDRESULT = 60 + SECOND(TIME1) - SECOND(TIME2).

    MINUTE(TIME2) is then incremented by 1.

    IF MINUTE(TIME2) <= MINUTE(TIME1)
    then MINUTERESULT = MINUTE(TIME1) - MINUTE(TIME2).

    IF MINUTE(TIME1) > MINUTE(TIME1)
    then MINUTERESULT = 60 + MINUTE(TIME1) - MINUTE(TIME2).

    HOUR(TIME2) is then incremented by 1.

    HOURRESULT = HOUR(TIME1) - HOUR(TIME2).

For example, the result of TIME('11:02:26') - '00:32:56' is 102930 (a duration of 10 hours, 29 minutes, and 30 seconds).

- The result of adding a duration to a time, or of subtracting a duration from a time, is itself a time. Any overflow or underflow of hours is discarded, thereby ensuring that the result is always a time. If a duration of hours is added or subtracted, only the hours portion of the time is affected. The minutes and seconds are unchanged.

  Similarly, if a duration of minutes is added or subtracted, only minutes and, if necessary, hours are affected. The seconds portion of the time is unchanged.

  Adding or subtracting a duration of seconds will, of course, affect the seconds portion of the time, and potentially the minutes and hours.

  Time durations, whether positive or negative, also can be added to and subtracted from times. The result is a time that has been incremented or decremented by the specified number of hours, minutes, and seconds, in that order. TIME1 + X, where X is a DECIMAL(6,0) number, is equivalent to the expression:

    TIME1 + HOUR(X) HOURS + MINUTE(X) MINUTES + SECOND(X) SECONDS

When subtracting a labeled duration of SECOND or SECONDS with a value that includes fractions of a second, the subtraction is performed as if the time value has up to 12 fractional second digits but the result is returned with the fractional seconds truncated.

Note: Although the time '24:00:00' is accepted as a valid time, it is never returned as the result of time addition or subtraction, even if the duration operand is zero (for example, time('24:00:00')±0 seconds = '00:00:00').

Timestamp arithmetic

Timestamps can be subtracted, incremented, or decremented.
**Datetime operations and durations**

- The result of subtracting one timestamp (TS2) from another (TS1) is a timestamp duration that specifies the number of years, months, days, hours, minutes, seconds, and fractional seconds between the two timestamps. The data type of the result is DECIMAL(14+s,s), where s is the maximum timestamp precision of TS1 and TS2.

  If TS1 is greater than or equal to TS2, TS2 is subtracted from TS1. If TS1 is less than TS2, however, TS1 is subtracted from TS2 and the sign of the result is made negative. The following procedural description clarifies the steps involved in the operation result = TS1 - TS2:

    If SECOND(TS2,s) <= SECOND(TS1,s) then SECOND(RESULT,s) = SECOND(TS1,s) - SECOND(TS2,s).
    If SECOND(TS2,s) > SECOND(TS1,s) then SECOND(RESULT,s) = 60 + SECOND(TS1,s) - SECOND(TS2,s).

    MINUTE(TS2) is then incremented by 1.

  The minutes part of the timestamps are subtracted as specified in the rules for subtracting times.

    If HOUR(TS2) <= HOUR(TS1) then HOUR(RESULT) = HOUR(TS1) - HOUR(TS2).
    If HOUR(TS2) > HOUR(TS1) then HOUR(RESULT) = 24 + HOUR(TS1) - HOUR(TS2)
    and DAY(TS2) is incremented by 1.

  The date part of the timestamps is subtracted as specified in the rules for subtracting dates.

- The result of subtracting a date (D1) from a timestamp (TS1) is the same as subtracting TIMESTAMP(D1) from TS1. Similarly, the result of subtracting one timestamp (TS1) from a date (D2) is the same as subtracting TS1 from TIMESTAMP(D2).

- The result of adding a duration to a timestamp, or of subtracting a duration from a timestamp is itself a timestamp. The precision of the result timestamp matches the precision of the timestamp operand. The date arithmetic portion is performed as previously defined, except that an overflow or underflow of hours is carried into the date part of the result, which must be within the range of valid dates. The time arithmetic portion is similar to time arithmetic except that it also considers the fractional seconds included in the duration. Thus, subtracting a duration, X, from a timestamp, TIMESTAMP1, where X is a DECIMAL(14+s,s) number, is equivalent to the expression:

    \[
    \text{TIMESTAMP1} - \text{YEAR}(X) \text{ YEARS} - \text{MONTH}(X) \text{ MONTHS} - \text{DAY}(X) \text{ DAYS} - \text{HOUR}(X) \text{ HOURS} - \text{MINUTE}(X) \text{ MINUTES} - \text{SECOND}(X, s) \text{ SECONDS}
    \]

  When subtracting a duration with non-zero scale or a labeled duration of SECONDS or SECONDS with a value that includes fractions of a second, the subtraction is performed as if the timestamp value has up to 12 fractional second digits. The resulting value is assigned to a timestamp value with the timestamp precision of the timestamp operand which could result in truncation of fractional second digits.
CASE expression

**case-expression:**

```
CASE searched-when-clause
    simple-when-clause
    ELSE NULL
    END
```

**searched-when-clause:**

```
WHEN search-condition THEN result-expression
```

**simple-when-clause:**

```
expression WHEN expression THEN result-expression
```

**Notes:**

1. If the result type of result-expression is a row type, then the syntax represents a row-case-expression and can only be used where a row-expression is allowed.

CASE expressions allow an expression to be selected based on the evaluation of one or more conditions. In general, the value of the case-expression is the value of the result-expression following the first (leftmost) case that evaluates to true. If no case evaluates to true and the ELSE keyword is present then the result is the value of the result-expression or NULL. If no case evaluates to true and the ELSE keyword is not present then the result is NULL. Note that when a case evaluates to unknown (because of NULLs), the case is not true and hence is treated the same way as a case that evaluates to false.

If the CASE expression is in a VALUES clause, an IN predicate, a GROUP BY clause, or an ORDER BY clause, the search-condition in a searched-when-clause cannot be a quantified predicate, IN predicate using a fullselect, or an EXISTS predicate (SQLSTATE 42625).

When using the simple-when-clause, the value of the expression prior to the first WHEN keyword is tested for equality with the value of the expression following the WHEN keyword. The data type of the expression prior to the first WHEN keyword must therefore be compatible to the data types of each expression following the WHEN keyword(s). The expression prior to the first WHEN keyword in a simple-when-clause cannot include a function that is not deterministic or has an external action (SQLSTATE 42845).

A result-expression is an expression following the THEN or ELSE keywords. There must be at least one result-expression in the CASE expression (NULL cannot be specified for every case) (SQLSTATE 42625). All result expressions must have compatible data types (SQLSTATE 42804).
CASE expression

Examples

- If the first character of a department number is a division in the organization, then a CASE expression can be used to list the full name of the division to which each employee belongs:

  ```sql
  SELECT EMPNO, LASTNAME,
  CASE SUBSTR(WORKDEPT,1,1)
  WHEN 'A' THEN 'Administration'
  WHEN 'B' THEN 'Human Resources'
  WHEN 'C' THEN 'Accounting'
  WHEN 'D' THEN 'Design'
  WHEN 'E' THEN 'Operations'
  END
  FROM EMPLOYEE;
  ```

- The number of years of education are used in the EMPLOYEE table to give the education level. A CASE expression can be used to group these and to show the level of education.

  ```sql
  SELECT EMPNO, FIRSTNME, MIDINIT, LASTNAME,
  CASE WHEN EDLEVEL < 15 THEN 'SECONDARY'
  WHEN EDLEVEL < 19 THEN 'COLLEGE'
  ELSE 'POST GRADUATE'
  END
  FROM EMPLOYEE
  ```

- Another interesting example of CASE statement usage is in protecting from division by 0 errors. For example, the following code finds the employees who earn more than 25% of their income from commission, but who are not fully paid on commission:

  ```sql
  SELECT EMPNO, WORKDEPT, SALARY+COMM FROM EMPLOYEE
  WHERE (CASE WHEN SALARY=0 THEN NULL
  ELSE COMM/SALARY
  END) > 0.25;
  ```

- The following CASE expressions are the same:

  ```sql
  SELECT LASTNAME,
  CASE WHEN LASTNAME = 'Haas' THEN 'President'
  ...

  SELECT LASTNAME,
  CASE LASTNAME
  WHEN 'Haas' THEN 'President'
  ...
  ```

There are two scalar functions, NULLIF and COALESCE, that are specialized to handle a subset of the functionality provided by CASE. Table 23 shows the equivalent expressions using CASE or these functions.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Equivalent Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>CASE WHEN e1=e2 THEN NULL ELSE e1 END</td>
<td>NULLIF(e1,e2)</td>
</tr>
<tr>
<td>CASE WHEN e1 IS NOT NULL THEN e1 ELSE e2 END</td>
<td>COALESCE(e1,e2)</td>
</tr>
</tbody>
</table>

Table 23. Equivalent CASE Expressions
### Table 23. Equivalent CASE Expressions (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Equivalent Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>CASE</td>
<td></td>
</tr>
<tr>
<td>WHEN e1 IS NOT NULL THEN e1</td>
<td>COALESCE(e1,e2,...,eN)</td>
</tr>
<tr>
<td>ELSE COALESCE(e2,...,eN)</td>
<td></td>
</tr>
<tr>
<td>END</td>
<td></td>
</tr>
<tr>
<td>CASE</td>
<td></td>
</tr>
<tr>
<td>WHEN c1=var1 OR (c1 IS NULL AND var1 IS NULL) THEN 'a'</td>
<td>DECODE(c1, var1, 'a', var2, 'b')</td>
</tr>
<tr>
<td>WHEN c1=var2 OR (c1 IS NULL AND var2 IS NULL) THEN 'b'</td>
<td></td>
</tr>
<tr>
<td>ELSE NULL</td>
<td></td>
</tr>
<tr>
<td>END</td>
<td></td>
</tr>
</tbody>
</table>
CAST specification

CAST specification

cast-specification:

```
CAST ( expression AS data-type NULL (1) parameter-marker
SCOPE typed-table-name typed-view-name
cursor-cast-specification
cursor-cast-specification
row-cast-specification
)
```

cursor-cast-specification:

```
parameter-marker AS CURSOR
cursor-type-name
```

row-cast-specification:

```
row-expression AS row-type-name
NULL
cursor-cast-specification
row-cast-specification

```

data-type:

```
built-in-type
distinct-type-name
structured-type-name
REF (type-name)
```

built-in-type:
### CAST specification

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMALLINT</td>
<td>32-bit signed integer</td>
<td>(5,0)</td>
</tr>
<tr>
<td>INTEGER</td>
<td>32-bit signed integer</td>
<td></td>
</tr>
<tr>
<td>INT</td>
<td>32-bit signed integer</td>
<td></td>
</tr>
<tr>
<td>BIGINT</td>
<td>64-bit signed integer</td>
<td></td>
</tr>
<tr>
<td>DECIMAL</td>
<td>Decimal with scale (integer, integer)</td>
<td></td>
</tr>
<tr>
<td>DEC</td>
<td>DECIMAL with scale (integer, integer)</td>
<td></td>
</tr>
<tr>
<td>NUMERIC</td>
<td>NUMERIC with scale (integer, integer)</td>
<td></td>
</tr>
<tr>
<td>NUM</td>
<td>NUMERIC with scale (integer, integer)</td>
<td></td>
</tr>
<tr>
<td>FLOAT</td>
<td>Floating point with precision (integer)</td>
<td>(34)</td>
</tr>
<tr>
<td>REAL</td>
<td>Floating point with precision (integer)</td>
<td></td>
</tr>
<tr>
<td>DOUBLE</td>
<td>Floating point with precision (integer)</td>
<td></td>
</tr>
<tr>
<td>DECIMAL</td>
<td>DECIMAL with scale (integer, integer)</td>
<td></td>
</tr>
<tr>
<td>DEC</td>
<td>DECIMAL with scale (integer, integer)</td>
<td></td>
</tr>
<tr>
<td>NUMERIC</td>
<td>NUMERIC with scale (integer, integer)</td>
<td></td>
</tr>
<tr>
<td>NUM</td>
<td>NUMERIC with scale (integer, integer)</td>
<td></td>
</tr>
<tr>
<td>FLOAT</td>
<td>Floating point with precision (integer)</td>
<td></td>
</tr>
<tr>
<td>REAL</td>
<td>Floating point with precision (integer)</td>
<td></td>
</tr>
<tr>
<td>DOUBLE</td>
<td>Floating point with precision (integer)</td>
<td></td>
</tr>
<tr>
<td>DECIMAL</td>
<td>DECIMAL with scale (integer, integer)</td>
<td></td>
</tr>
<tr>
<td>DEC</td>
<td>DECIMAL with scale (integer, integer)</td>
<td></td>
</tr>
<tr>
<td>NUMERIC</td>
<td>NUMERIC with scale (integer, integer)</td>
<td></td>
</tr>
<tr>
<td>NUM</td>
<td>NUMERIC with scale (integer, integer)</td>
<td></td>
</tr>
<tr>
<td>FLOAT</td>
<td>Floating point with precision (integer)</td>
<td></td>
</tr>
<tr>
<td>REAL</td>
<td>Floating point with precision (integer)</td>
<td></td>
</tr>
<tr>
<td>DOUBLE</td>
<td>Floating point with precision (integer)</td>
<td></td>
</tr>
<tr>
<td>DECIMAL</td>
<td>DECIMAL with scale (integer, integer)</td>
<td></td>
</tr>
<tr>
<td>DEC</td>
<td>DECIMAL with scale (integer, integer)</td>
<td></td>
</tr>
<tr>
<td>NUMERIC</td>
<td>NUMERIC with scale (integer, integer)</td>
<td></td>
</tr>
<tr>
<td>NUM</td>
<td>NUMERIC with scale (integer, integer)</td>
<td></td>
</tr>
<tr>
<td>FLOAT</td>
<td>Floating point with precision (integer)</td>
<td></td>
</tr>
<tr>
<td>REAL</td>
<td>Floating point with precision (integer)</td>
<td></td>
</tr>
<tr>
<td>DOUBLE</td>
<td>Floating point with precision (integer)</td>
<td></td>
</tr>
<tr>
<td>DECIMAL</td>
<td>DECIMAL with scale (integer, integer)</td>
<td></td>
</tr>
<tr>
<td>DEC</td>
<td>DECIMAL with scale (integer, integer)</td>
<td></td>
</tr>
<tr>
<td>NUMERIC</td>
<td>NUMERIC with scale (integer, integer)</td>
<td></td>
</tr>
<tr>
<td>NUM</td>
<td>NUMERIC with scale (integer, integer)</td>
<td></td>
</tr>
<tr>
<td>FLOAT</td>
<td>Floating point with precision (integer)</td>
<td></td>
</tr>
<tr>
<td>REAL</td>
<td>Floating point with precision (integer)</td>
<td></td>
</tr>
<tr>
<td>DOUBLE</td>
<td>Floating point with precision (integer)</td>
<td></td>
</tr>
<tr>
<td>DECIMAL</td>
<td>DECIMAL with scale (integer, integer)</td>
<td></td>
</tr>
<tr>
<td>DEC</td>
<td>DECIMAL with scale (integer, integer)</td>
<td></td>
</tr>
<tr>
<td>NUMERIC</td>
<td>NUMERIC with scale (integer, integer)</td>
<td></td>
</tr>
<tr>
<td>NUM</td>
<td>NUMERIC with scale (integer, integer)</td>
<td></td>
</tr>
<tr>
<td>FLOAT</td>
<td>Floating point with precision (integer)</td>
<td></td>
</tr>
<tr>
<td>REAL</td>
<td>Floating point with precision (integer)</td>
<td></td>
</tr>
<tr>
<td>DOUBLE</td>
<td>Floating point with precision (integer)</td>
<td></td>
</tr>
<tr>
<td>DECIMAL</td>
<td>DECIMAL with scale (integer, integer)</td>
<td></td>
</tr>
<tr>
<td>DEC</td>
<td>DECIMAL with scale (integer, integer)</td>
<td></td>
</tr>
<tr>
<td>NUMERIC</td>
<td>NUMERIC with scale (integer, integer)</td>
<td></td>
</tr>
<tr>
<td>NUM</td>
<td>NUMERIC with scale (integer, integer)</td>
<td></td>
</tr>
<tr>
<td>FLOAT</td>
<td>Floating point with precision (integer)</td>
<td></td>
</tr>
<tr>
<td>REAL</td>
<td>Floating point with precision (integer)</td>
<td></td>
</tr>
<tr>
<td>DOUBLE</td>
<td>Floating point with precision (integer)</td>
<td></td>
</tr>
<tr>
<td>DECIMAL</td>
<td>DECIMAL with scale (integer, integer)</td>
<td></td>
</tr>
<tr>
<td>DEC</td>
<td>DECIMAL with scale (integer, integer)</td>
<td></td>
</tr>
<tr>
<td>NUMERIC</td>
<td>NUMERIC with scale (integer, integer)</td>
<td></td>
</tr>
<tr>
<td>NUM</td>
<td>NUMERIC with scale (integer, integer)</td>
<td></td>
</tr>
<tr>
<td>FLOAT</td>
<td>Floating point with precision (integer)</td>
<td></td>
</tr>
<tr>
<td>REAL</td>
<td>Floating point with precision (integer)</td>
<td></td>
</tr>
<tr>
<td>DOUBLE</td>
<td>Floating point with precision (integer)</td>
<td></td>
</tr>
<tr>
<td>DECIMAL</td>
<td>DECIMAL with scale (integer, integer)</td>
<td></td>
</tr>
<tr>
<td>DEC</td>
<td>DECIMAL with scale (integer, integer)</td>
<td></td>
</tr>
<tr>
<td>NUMERIC</td>
<td>NUMERIC with scale (integer, integer)</td>
<td></td>
</tr>
<tr>
<td>NUM</td>
<td>NUMERIC with scale (integer, integer)</td>
<td></td>
</tr>
<tr>
<td>FLOAT</td>
<td>Floating point with precision (integer)</td>
<td></td>
</tr>
<tr>
<td>REAL</td>
<td>Floating point with precision (integer)</td>
<td></td>
</tr>
<tr>
<td>DOUBLE</td>
<td>Floating point with precision (integer)</td>
<td></td>
</tr>
<tr>
<td>DECIMAL</td>
<td>DECIMAL with scale (integer, integer)</td>
<td></td>
</tr>
<tr>
<td>DEC</td>
<td>DECIMAL with scale (integer, integer)</td>
<td></td>
</tr>
<tr>
<td>NUMERIC</td>
<td>NUMERIC with scale (integer, integer)</td>
<td></td>
</tr>
<tr>
<td>NUM</td>
<td>NUMERIC with scale (integer, integer)</td>
<td></td>
</tr>
<tr>
<td>FLOAT</td>
<td>Floating point with precision (integer)</td>
<td></td>
</tr>
<tr>
<td>REAL</td>
<td>Floating point with precision (integer)</td>
<td></td>
</tr>
<tr>
<td>DOUBLE</td>
<td>Floating point with precision (integer)</td>
<td></td>
</tr>
<tr>
<td>DECIMAL</td>
<td>DECIMAL with scale (integer, integer)</td>
<td></td>
</tr>
<tr>
<td>DEC</td>
<td>DECIMAL with scale (integer, integer)</td>
<td></td>
</tr>
<tr>
<td>NUMERIC</td>
<td>NUMERIC with scale (integer, integer)</td>
<td></td>
</tr>
<tr>
<td>NUM</td>
<td>NUMERIC with scale (integer, integer)</td>
<td></td>
</tr>
<tr>
<td>FLOAT</td>
<td>Floating point with precision (integer)</td>
<td></td>
</tr>
<tr>
<td>REAL</td>
<td>Floating point with precision (integer)</td>
<td></td>
</tr>
<tr>
<td>DOUBLE</td>
<td>Floating point with precision (integer)</td>
<td></td>
</tr>
<tr>
<td>DECIMAL</td>
<td>DECIMAL with scale (integer, integer)</td>
<td></td>
</tr>
<tr>
<td>DEC</td>
<td>DECIMAL with scale (integer, integer)</td>
<td></td>
</tr>
<tr>
<td>NUMERIC</td>
<td>NUMERIC with scale (integer, integer)</td>
<td></td>
</tr>
<tr>
<td>NUM</td>
<td>NUMERIC with scale (integer, integer)</td>
<td></td>
</tr>
</tbody>
</table>
CAST specification

The CAST specification returns the cast operand (the first operand) cast to the type specified by the data-type. If the cast is not supported, an error is returned (SQLSTATE 42846).

expression

If the cast operand is an expression (other than parameter marker or NULL), the result is the argument value converted to the specified target data-type.

When casting character strings (other than CLOBs) to a character string with a different length, a warning (SQLSTATE 01004) is returned if truncation of other than trailing blanks occurs. When casting graphic character strings (other than DBCLOBs) to a graphic character string with a different length, a warning (SQLSTATE 01004) is returned if truncation of other than trailing blanks occurs. For BLOB, CLOB and DBCLOB cast operands, the warning is issued if any characters are truncated.

When casting an array, the target data type must be a user-defined array data type (SQLSTATE 42821). The data type of the elements of the array must be the same as the data type of the elements of the target array data type (SQLSTATE 42846). The cardinality of the array must be less than or equal to the maximum cardinality of the target array data type (SQLSTATE 2202F).

NULL

If the cast operand is the keyword NULL, the result is a null value that has the specified data-type.

parameter-marker

A parameter marker is normally considered an expression, but is documented separately in this case because it has a special meaning. If the cast operand is a parameter-marker, the specified data-type is considered a promise that the replacement will be assignable to the specified data type (using store assignment for strings). Such a parameter marker is considered a typed parameter marker. Typed parameter markers will be treated like any other typed value for the purpose of function resolution, DESCRIBE of a select list or for column assignment.

cursor-cast-specification

A cast specification used to indicate that a parameter marker is expected to be a cursor type. It can be used wherever an expression is supported in contexts that allow cursor types.

parameter-marker

The cast operand is a parameter marker and is considered a promise that the replacement will be assignable to the specified cursor type.

CURSOR

Specifies the built-in data type CURSOR.

cursor-type-name

Specifies the name of a user-defined cursor type.

row-cast-specification

A cast specification where the input is a row value and the result is a user-defined row type. A row-cast-specification is only valid where a row-expression is allowed.

row-expression

The data type of row-expression must be a variable of row type that is anchored to the definition of a table or view. The data type of row-expression must not be a user-defined row type (SQLSTATE 42846).
CAST specification

NULL
Specifies that the cast operand is the null value. The result is a row with the null value for every field of the specified data type.

parameter-marker
The cast operand is a parameter marker and is considered a promise that the replacement will be assignable to the specified row-type-name.

row-type-name
Specifies the name of a user-defined row type. The row-expression must be castable to row-type-name (SQLSTATE 42846).

data-type
The name of an existing data type. If the type name is not qualified, the SQL path is used to perform data type resolution. A data type that has associated attributes, such as length or precision and scale, should include these attributes when specifying data-type. (CHAR defaults to a length of 1, DECIMAL defaults to a precision of 5 and a scale of 0, and DECFLOAT defaults to a precision of 34 if not specified.) The FOR SBCS DATA clause or the FOR MIXED DATA clause (only one is supported depending on whether or not the database supports the graphic data type) can be used to cast a FOR BIT DATA string to the database code page. Restrictions on the supported data types are based on the specified cast operand.

- For a cast operand that is an expression, the supported target data types depend on the data type of the cast operand (source data type).
- For a cast operand that is the keyword NULL, any existing data type can be used.
- For a cast operand that is a parameter marker, the target data type can be any existing data type. If the data type is a user-defined distinct type, the application using the parameter marker will use the source data type of the user-defined distinct type. If the data type is a user-defined structured type, the application using the parameter marker will use the input parameter type of the TO SQL transform function for the user-defined structured type.

SCOPE
When the data type is a reference type, a scope may be defined that identifies the target table or target view of the reference.

typed-table-name
The name of a typed table. The table must already exist (SQLSTATE 42704). The cast must be to data-type REF(S), where S is the type of typed-table-name (SQLSTATE 428DM).

typed-view-name
The name of a typed view. The view must exist or have the same name as the view being created that includes the cast as part of the view definition (SQLSTATE 42704). The cast must be to data-type REF(S), where S is the type of typed-view-name (SQLSTATE 428DM).

When numeric data is cast to character data, the result data type is a fixed-length character string. When character data is cast to numeric data, the result data type depends on the type of number specified. For example, if cast to integer, it becomes a large integer.

Examples
- An application is only interested in the integer portion of the SALARY (defined as decimal(9,2)) from the EMPLOYEE table. The following query, including the employee number and the integer value of SALARY, could be prepared.
**CAST specification**

```sql
SELECT EMPNO, CAST(SALARY AS INTEGER) FROM EMPLOYEE
```

- Assume the existence of a distinct type called T_AGE that is defined on SMALLINT and used to create column AGE in PERSONNEL table. Also assume the existence of a distinct type called R_YEAR that is defined on INTEGER and used to create column RETIRE_YEAR in PERSONNEL table. The following update statement could be prepared.

```sql
UPDATE PERSONNEL SET RETIRE_YEAR =?
WHERE AGE = CAST( ? AS T_AGE)
```

The first parameter is an untyped parameter marker that would have a data type of R_YEAR, although the application will use an integer for this parameter marker. This does not require the explicit CAST specification because it is an assignment.

The second parameter marker is a typed parameter marker that is cast as a distinct type T_AGE. This satisfies the requirement that the comparison must be performed with compatible data types. The application will use the source data type (which is SMALLINT) for processing this parameter marker.

Successful processing of this statement assumes that the SQL path includes the schema name of the schema (or schemas) where the two distinct types are defined.

- An application supplies a value that is a series of bits, for example an audio stream, and it should not undergo code page conversion before being used in an SQL statement. The application could use the following CAST:

```sql
CAST( ? AS VARCHAR(10000) FOR BIT DATA)
```

- Assume that an array type and a table have been created as follows:

```sql
CREATE TYPE PHONELIST AS DECIMAL(10, 0) ARRAY[5]
CREATE TABLE EMP_PHONES
(ID INTEGER,
 PHONENUMBER DECIMAL(10,0) );
```

The following procedure returns an array with the phone numbers for the employee with ID 1775. If there are more than five phone numbers for this employee, an error is returned (SQLSTATE 2202F).

```sql
CREATE PROCEDURE GET_PHONES(OUT EPHONES PHONELIST)
BEGIN
SELECT CAST(ARRAY_AGG(PHONENUMBER) AS PHONELIST)
INTO EPHONES
FROM EMP_PHONES
WHERE ID = 1775;
END
```
Field reference

Field reference

field-reference:

[\textnormal{row-variable-name} \cdot \textnormal{field-name}]

[\textnormal{row-array-element-specification} \cdot \textnormal{field-name}]

A field of a row type is referenced using the field name qualified by:

- A variable that returns a row type which includes a field with that field name
- An array element specification that returns a row type which includes a field with that field name

\texttt{row-variable-name}

The name of a variable with a data type that is a row type.

\texttt{row-array-element-specification}

An \texttt{array-element-specification} where the data type of the array element is a row type.

\texttt{field-name}

The name of a field within the row type.
The XMLCAST specification returns the cast operand (the first operand) cast to the type specified by the data type. XMLCAST supports casts involving XML values, including conversions between non-XML data types and the XML data type. If the cast is not supported, an error is returned (SQLSTATE 22003).

**expression**

If the cast operand is an expression (other than a parameter marker or NULL), the result is the argument value converted to the specified target data type. The expression or the target data type must be the XML data type (SQLSTATE 42846).

**NULL**

If the cast operand is the keyword NULL, the target data type must be the XML data type (SQLSTATE 42846). The result is a null XML value.

**parameter-marker**

If the cast operand is a parameter marker, the target data type must be XML (SQLSTATE 42846). A parameter marker is normally considered to be an expression, but is documented separately in this case because it has special meaning. If the cast operand is a parameter marker, the specified data type is considered to be a promise that the replacement will be assignable to the specified (XML) data type (using store assignment). Such a parameter marker is considered to be a typed parameter marker, which is treated like any other typed value for the purpose of function resolution, a describe operation on a select list, or column assignment.

**data-type**

The name of an existing SQL data type. If the name is not qualified, the SQL path is used to perform data type resolution. If a data type has associated attributes, such as length or precision and scale, these attributes should be included when specifying a value for *data-type*. CHAR defaults to a length of 1, and DECIMAL defaults to a precision of 5 and a scale of 0 if not specified. Restrictions on the supported data types are based on the specified cast operand.

- For a cast operand that is an expression, the supported target data types depend on the data type of the cast operand (source data type).
- For a cast operand that is the keyword NULL, the target data type must be XML.
- For a cast operand that is a parameter marker, the target data type must be XML.

**Note: Support in non-Unicode databases:** When XMLCAST is used to convert an XML value to an SQL data type, code page conversion is performed. The encoding of the cast expression is converted from UTF-8 to the database code page. Characters in the original expression that are not present in the database code page are replaced by substitution characters as a result of this conversion.
Examples

• Create a null XML value.
  
  `XMLCAST(NULL AS XML)`

• Convert a value extracted from an XMLQUERY expression into an INTEGER:
  
  `XMLCAST(XMLQUERY('m/PRODUCT/QUANTITY'
                     PASSING BY REF xmlcol AS "m" RETURNING SEQUENCE) AS INTEGER)`

• Convert a value extracted from an XMLQUERY expression into a varying-length character string:
  
  `XMLCAST(XMLQUERY('m/PRODUCT/ADD-TIMESTAMP'
                     PASSING BY REF xmlcol AS "m" RETURNING SEQUENCE) AS VARCHAR(30))`

• Convert a value extracted from an SQL scalar subquery into an XML value.
  
  `XMLCAST((SELECT quantity FROM product AS p
              WHERE p.id = 1077) AS XML)`
ARRAY element specification

**ARRAY element specification**

array-element-specification:

```
| array-variable | CAST (parameter-marker AS data-type) | [expression] |
```

Notes:

1. If the data type of the elements in the array is a row type, then the syntax represents an array-element-specification with a row data type and can only be used where a row-expression is allowed.

The ARRAY element specification returns the element from an array specified by expression. If any argument to expression is null, the null value is returned.

- **array-variable**
  - An SQL variable, SQL parameter, or global variable of an array type.

- **CAST (parameter-marker AS data-type)**
  - A parameter marker is normally considered to be an expression, but in this case it must explicitly be cast to a user-defined array data type.

- **[expression]**
  - Specifies the array index of the element that is to be extracted from the array. The array index of an ordinary array must be assignable to INTEGER (SQLSTATE 428H1); its value must be between 1 and the cardinality of the array (SQLSTATE 2202E). The array index of an associative array must be assignable to the index data type (SQLSTATE 428H1).
Array constructor

An array constructor is a language element that can be used to define and construct an array data type value within a valid context.

Syntax

```
ARRAY [ fullselect ]
```

Authorization

No specific authorizations are required to reference an array constructor within an SQL statement, however for the statement execution to be successful all other authorization requirements for the statement must be satisfied.

Description

**ARRAY[]**

Specifies an empty array.

**ARRAY[fullselect]**

Specifies an array whose elements are the result rows of a fullselect that returns a single column.

If the fullselect includes an order-by-clause, the order determines the order in which row values are assigned to elements of the array. If no order-by-clause is specified, the order in which row values are assigned to elements of the array is non deterministic.

**ARRAY[element-expression,...]**

Specifies an array using the value of a an expression for each element. The first element-expression is assigned to the array element with array index 1. The second element-expression is assigned to the array element with array index 2 and so on. Every element-expression must have a compatible data type with every other element-expression, where the base type of the array is determined based on the “Rules for result data types”.

Rules

- The base type of the array-constructor, as derived from the element-expressions or the fullselect, must be assignable to the base type of the target array (SQLSTATE 42821).
- The number of elements in the array-constructor must not exceed the maximum cardinality of the target array variable (SQLSTATE 2202F).

Notes

- An array constructor can be used to define only an ordinary array with elements that are not a row type. An array constructor cannot be used to define an associative array or an ordinary array with elements that are a row type. Such arrays can only be constructed by assigning the individual elements.
Array constructor

Examples

Example 1: Set the array variable RECENT_CALLS of array type PHONENUMBERS to an array of fixed numbers.

```sql
SET RECENT_CALLS = ARRAY[9055553907, 4165554213, 4085553678]
```

Example 2: Set the array variable DEPT_PHONES of array type PHONENUMBERS to an array of phone numbers retrieved from the DEPARTMENT_INFO table.

```sql
SET DEPT_PHONES = ARRAY(SELECT DECIMAL(AREA_CODE CONCAT '555' CONCAT EXTENSION,16) FROM DEPARTMENT_INFO WHERE DEPTID = 624)
```
**Dereference operation**

**dereference-operation:**

```
  scoped-ref-expression  -->  name1
```

The scope of the scoped reference expression is a table or view called the *target* table or view. The scoped reference expression identifies a *target row*. The *target row* is the row in the target table or view (or in one of its subtables or subviews) whose object identifier (OID) column value matches the reference expression. The dereference operation can be used to access a column of the target row, or to invoke a method, using the target row as the subject of the method. The result of a dereference operation can always be null. The dereference operation takes precedence over all other operators.

**scoped-ref-expression**

An expression that is a reference type that has a scope (SQLSTATE 428DT). If the expression is a host variable, parameter marker or other unscoped reference type value, a `CAST` specification with a `SCOPE` clause is required to give the reference a scope.

**name1**

Specifies an unqualified identifier.

If no parentheses follow `name1`, and `name1` matches the name of an attribute of the target type, then the value of the dereference operation is the value of the named column in the target row. In this case, the data type of the column (made nullable) determines the result type of the dereference operation. If no target row exists whose object identifier matches the reference expression, then the result of the dereference operation is null. If the dereference operation is used in a select list and is not included as part of an expression, `name1` becomes the result column name.

If parentheses follow `name1`, or if `name1` does not match the name of an attribute of the target type, then the dereference operation is treated as a method invocation. The name of the invoked method is `name1`. The subject of the method is the target row, considered as an instance of its structured type. If no target row exists whose object identifier matches the reference expression, the subject of the method is a null value of the target type. The expressions inside parentheses, if any, provide the remaining parameters of the method invocation. The normal process is used for resolution of the method invocation. The result type of the selected method (made nullable) determines the result type of the dereference operation.

The authorization ID of the statement that uses a dereference operation must have SELECT privilege on the target table of the `scoped-ref-expression` (SQLSTATE 42501).

A dereference operation can never modify values in the database. If a dereference operation is used to invoke a mutator method, the mutator method modifies a copy of the target row and returns the copy, leaving the database unchanged.
Dereference operation

Examples

- Assume the existence of an EMPLOYEE table that contains a column called DEPTREF which is a reference type scoped to a typed table based on a type that includes the attribute DEPTNAME. The values of DEPTREF in the table EMPLOYEE should correspond to the OID column values in the target table of DEPTREF column.

  ```sql
  SELECT EMPNO, DEPTREF->DEPTNAME
  FROM EMPLOYEE
  ```

- Using the same tables as in the previous example, use a dereference operation to invoke a method named BUDGET, with the target row as subject parameter, and '1997' as an additional parameter.

  ```sql
  SELECT EMPNO, DEPTREF->BUDGET('1997') AS DEPTBUDGET97
  FROM EMPLOYEE
  ```
Both system-generated observer and mutator methods, as well as user-defined methods are invoked using the double-dot operator.

subject-expression
An expression with a static result type that is a user-defined structured type.

method-name
The unqualified name of a method. The static type of subject-expression or one of its supertypes must include a method with the specified name.

(expression,...)
The arguments of method-name are specified within parentheses. Empty parentheses can be used to indicate that there are no arguments. The method-name and the data types of the specified argument expressions are used to resolve to the specific method, based on the static type of subject-expression.

The double-dot operator used for method invocation is a high precedence left to right infix operator. For example, the following two expressions are equivalent:

\[ \text{a..b..c + x..y..z} \]

and

\[ ((\text{a..b})..\text{c}) + ((\text{x..y})..\text{z}) \]

If a method has no parameters other than its subject, it can be invoked with or without parentheses. For example, the following two expressions are equivalent:

\[ \text{point1..x} \]
\[ \text{point1..x()} \]

Null subjects in method calls are handled as follows:

- If a system-generated mutator method is invoked with a null subject, an error results (SQLSTATE 2202D)
- If any method other than a system-generated mutator is invoked with a null subject, the method is not executed, and its result is null. This rule includes user-defined methods with SELF AS RESULT.

When a database object (a package, view, or trigger, for example) is created, the best fit method that exists for each of its method invocations is found.

Note: Methods of types defined WITH FUNCTION ACCESS can also be invoked using the regular function notation. Function resolution considers all functions, as well as methods with function access as candidate functions. However, functions cannot be invoked using method invocation. Method resolution considers all methods and does not consider functions as candidate methods. Failure to resolve to an appropriate function or method results in an error (SQLSTATE 42884).
Method invocation

**Example**

- Use the double-dot operator to invoke a method called `AREA`. Assume the existence of a table called `RINGS`, with a column `CIRCLE_COL` of structured type `CIRCLE`. Also, assume that the method `AREA` has been defined previously for the `CIRCLE` type as `AREA()` RETURNS `DOUBLE`.

  ```sql
  SELECT CIRCLE_COL..AREA() FROM RINGS
  ```
OLAP specifications

OLAP specifications

OLAP-specification:

ordered-OLAP-specification:

numbering-specification

aggregation-specification

lag-function:

LEAD (expression)

OVER (window-partition-clause)

window-order-clause:

PARTITION BY partitioning-expression

lead-function:

LAG (expression)

OVER (window-partition-clause)

window-order-clause:

ORDER BY sort-key-expression asc-option
desc-option

ORDER OF table-designator

window-partition-clause:
OLAP specifications

asc-option:

ASC | NULLS LAST | NULLS FIRST

desc-option:

DESC | NULLS FIRST | NULLS LAST

numbering-specification:

ROW_NUMBER () OVER (window-partition-clause)

window-order-clause

aggregation-specification:

(1) column-function OLAP-column-function OVER (window-partition-clause)

RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING

window-order-clause RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW

window-aggregation-group-clause

OLAP-column-function:

first-value-function

last-value-function

first-value-function:

FIRST_VALUE (expression RESPECT NULLS) IGNORE NULLS

last-value-function:

LAST_VALUE (expression RESPECT NULLS) IGNORE NULLS
OLAP specifications

window-aggregation-group-clause:

\[
\text{ROWS} \quad \text{RANGE} \\
\text{group-start} \quad \text{group-between} \quad \text{group-end}
\]

\[
\text{group-start:} \\
\text{UNBOUNDED PRECEDING} \\
\text{unsigned-constant} \quad \text{PRECEDING} \\
\text{CURRENT ROW}
\]

\[
\text{group-between:} \\
\text{BETWEEN} \quad \text{group-bound1} \quad \text{AND} \quad \text{group-bound2}
\]

\[
\text{group-bound1:} \\
\text{UNBOUNDED PRECEDING} \\
\text{unsigned-constant} \quad \text{PRECEDING} \\
\text{unsigned-constant} \quad \text{FOLLOWING} \\
\text{CURRENT ROW}
\]

\[
\text{group-bound2:} \\
\text{UNBOUNDED FOLLOWING} \\
\text{unsigned-constant} \quad \text{PRECEDING} \\
\text{unsigned-constant} \quad \text{FOLLOWING} \\
\text{CURRENT ROW}
\]

\[
\text{group-end:} \\
\text{UNBOUNDED FOLLOWING} \\
\text{unsigned-constant} \quad \text{FOLLOWING}
\]

Notes:

1. ARRAY\_AGG is not supported as an aggregate function in aggregation-specification (SQLSTATE 42887).

On-Line Analytical Processing (OLAP) functions provide the ability to return ranking, row numbering and existing aggregate function information as a scalar value in a query result. An OLAP function can be included in expressions in a select-list or the ORDER BY clause of a select-statement (SQLSTATE 42903). An OLAP function cannot be used within an argument to an XMLQUERY or XMLNOTES expression (SQLSTATE 42903). An OLAP function cannot be used as an argument of an aggregate function (SQLSTATE 42607). The query result to which the OLAP function is applied is the result table of the innermost subselect that includes the OLAP function.

When specifying an OLAP function, a window is specified that defines the rows over which the function is applied, and in what order. When used with an aggregate function, the applicable rows can be further refined, relative to the
The ranking function computes the ordinal rank of a row within the window. Rows that are not distinct with respect to the ordering within their window are assigned the same rank. The results of ranking may be defined with or without gaps in the numbers resulting from duplicate values.

If RANK is specified, the rank of a row is defined as 1 plus the number of rows that strictly precede the row. Thus, if two or more rows are not distinct with respect to the ordering, then there will be one or more gaps in the sequential rank numbering.

If DENSE_RANK (or DENSERANK) is specified, the rank of a row is defined as 1 plus the number of preceding rows that are distinct with respect to the ordering. Therefore, there will be no gaps in the sequential rank numbering.

The ROW_NUMBER (or ROWNUMBER) function computes the sequential row number of the row within the window defined by the ordering, starting with 1 for the first row. If the ORDER BY clause is not specified in the window, the row numbers are assigned to the rows in arbitrary order, as returned by the subselect (not according to any ORDER BY clause in the select-statement).

If the FETCH FIRST n ROWS ONLY clause is used along with the ROW_NUMBER function, the row numbers might not be displayed in order. The FETCH FIRST clause is applied after the result set (including any ROW_NUMBER assignments) is generated; therefore, if the row number order is not the same as the order of the result set, some assigned numbers might be missing from the sequence.

The data type of the result of RANK, DENSE_RANK or ROW_NUMBER is BIGINT. The result cannot be null.

The LAG function returns the expression value for the row at offset rows before the current row. The offset must be a positive integer (SQLSTATE 42815). An offset value of 0 means the current row. If a window-partition-clause is specified, offset means offset rows before the current row and within the current partition. If offset is not specified, the value 1 is used. If default-value (which can be an expression) is specified, it will be returned if the offset goes beyond the scope of the current partition. Otherwise, the null value is returned. If 'IGNORE NULLS' is specified, all rows where the expression value for the row is the null value are not considered in the calculation. If 'IGNORE NULLS' is specified and all rows are null, default-value (or the null value if default-value was not specified) is returned.

The LEAD function returns the expression value for the row at offset rows after the current row. The offset must be a positive integer (SQLSTATE 42815). An offset value of 0 means the current row. If a window-partition-clause is specified, offset means offset rows after the current row and within the current partition. If offset is not specified, the value 1 is used. If default-value (which can be an expression) is specified, it will be returned if the offset goes beyond the scope of the current partition. Otherwise, the null value is returned. If 'IGNORE NULLS' is specified, all rows where the expression value for the row is the null value are not considered in the calculation. If 'IGNORE NULLS' is specified and all rows are null, default-value (or the null value if default-value was not specified) is returned.
The **FIRST_VALUE** function returns the expression value for the first row in an OLAP window. If 'IGNORE NULLS' is specified, all rows where the expression value for the row is the null value are not considered in the calculation. If 'IGNORE NULLS' is specified and all values in the OLAP window are null, **FIRST_VALUE** returns the null value.

The **LAST_VALUE** function returns the expression value for the last row in an OLAP window. If 'IGNORE NULLS' is specified, all rows where the expression value for the row is the null value are not considered in the calculation. If 'IGNORE NULLS' is specified and all values in the OLAP window are null, **LAST_VALUE** returns the null value.

The data type of the result of **FIRST_VALUE**, **LAG**, **LAST_VALUE**, and **LEAD** is the data type of the expression. The result can be null.

**PARTITION BY** (*partitioning-expression*,*)

Defines the partition within which the function is applied. A *partitioning-expression* is an expression that is used in defining the partitioning of the result set. Each *column-name* that is referenced in a *partitioning-expression* must unambiguously reference a column of the result table of the subselect that contains the OLAP specification (SQLSTATE 42702 or 42703). A *partitioning-expression* cannot include a scalar fullselect or an XMLQUERY or XMLEXISTS expression (SQLSTATE 42822), or any function or query that is not deterministic or that has an external action (SQLSTATE 42845).

**window-order-clause**

**ORDER BY** (*sort-key-expression*,*)

Defines the ordering of rows within a partition that determines the value of the OLAP function or the meaning of the ROW values in the window-aggregation-group-clause (it does not define the ordering of the query result set).

*sort-key-expression*

An expression used in defining the ordering of the rows within a window partition. Each column name referenced in a *sort-key-expression* must unambiguously reference a column of the result set of the subselect, including the OLAP function (SQLSTATE 42702 or 42703). A *sort-key-expression* cannot include a scalar fullselect or an XMLQUERY or XMLEXISTS expression (SQLSTATE 42822), or any function or query that is not deterministic or that has an external action (SQLSTATE 42845). This clause is required for the **RANK** and **DENSE_RANK** functions (SQLSTATE 42601).

**ASC**

Uses the values of the *sort-key-expression* in ascending order.

**DESC**

Uses the values of the *sort-key-expression* in descending order.

**NULLS FIRST**

The window ordering considers null values *before* all non-null values in the sort order.

**NULLS LAST**

The window ordering considers null values *after* all non-null values in the sort order.

**ORDER OF** *table-designator*

Specifies that the same ordering used in *table-designator* should be applied
OLAP specifications

to the result table of the subselect. There must be a table reference
matching table-designator in the FROM clause of the subselect that specifies
this clause (SQLSTATE 42703). The subselect (or fullselect) corresponding
to the specified table-designator must include an ORDER BY clause that is
dependant on the data (SQLSTATE 428FI). The ordering that is applied is
the same as if the columns of the ORDER BY clause in the nested subselect
(or fullselect) were included in the outer subselect (or fullselect), and these
columns were specified in place of the ORDER OF clause.

window-aggregation-group-clause
The aggregation group of a row R is a set of rows defined in relation to R (in
the ordering of the rows of R’s partition). This clause specifies the aggregation
group. If this clause is not specified and a window-order-clause is also not
specified, the aggregation group consists of all the rows of the window
partition. This default can be specified explicitly using RANGE (as shown) or
ROWS.

If window-order-clause is specified, the default behavior is different when
window-aggregation-group-clause is not specified. The window aggregation
group consists of all rows of the partition of R that precede R or that are peers
of R in the window ordering of the window partition defined by the
window-order-clause.

ROWS
Indicates the aggregation group is defined by counting rows.

RANGE
Indicates the aggregation group is defined by an offset from a sort key.

group-start
Specifies the starting point for the aggregation group. The aggregation
group end is the current row. Specification of the group-start clause is
equivalent to a group-between clause of the form "BETWEEN group-start
AND CURRENT ROW".

group-between
Specifies the aggregation group start and end based on either ROWS or
RANGE.

group-end
Specifies the ending point for the aggregation group. The aggregation
group start is the current row. Specification of the group-end clause is
equivalent to a group-between clause of the form "BETWEEN CURRENT
ROW AND group-end".

UNBOUNDED PRECEDING
Includes the entire partition preceding the current row. This can be
specified with either ROWS or RANGE. Also, this can be specified with
multiple sort-key-expressions in the window-order-clause.

UNBOUNDED FOLLOWING
Includes the entire partition following the current row. This can be
specified with either ROWS or RANGE. Also, this can be specified with
multiple sort-key-expressions in the window-order-clause.

CURRENT ROW
Specifies the start or end of the aggregation group based on the current
row. If ROWS is specified, the current row is the aggregation group
boundary. If RANGE is specified, the aggregation group boundary includes
OLAP specifications

the set of rows with the same values for the sort-key-expressions as the current row. This clause cannot be specified in group-bound2 if group-bound1 specifies value FOLLOWING.

value PRECEDING
Specifications either the range or number of rows preceding the current row. If ROWS is specified, then value is a positive integer indicating a number of rows. If RANGE is specified, then the data type of value must be comparable to the type of the sort-key-expression of the window-order-clause. There can only be one sort-key-expression, and the data type of the sort-key-expression must allow subtraction. This clause cannot be specified in group-bound2 if group-bound1 is CURRENT ROW or value FOLLOWING.

value FOLLOWING
Specifications either the range or number of rows following the current row. If ROWS is specified, then value is a positive integer indicating a number of rows. If RANGE is specified, then the data type of value must be comparable to the type of the sort-key-expression of the window-order-clause. There can only be one sort-key-expression, and the data type of the sort-key-expression must allow addition.

Examples

- Display the ranking of employees, in order by surname, according to their total salary (based on salary plus bonus) that have a total salary more than $30,000.

  ```sql
  SELECT EMPNO, LASTNAME, FIRSTNAME, SALARY+BONUS AS TOTAL_SALARY,
  RANK() OVER (ORDER BY SALARY+BONUS DESC) AS RANK_SALARY
  FROM EMPLOYEE
  WHERE SALARY+BONUS > 30000
  ORDER BY LASTNAME
  ```

  Note that if the result is to be ordered by the ranking, then replace ORDER BY LASTNAME with:

  ```sql
  ORDER BY RANK_SALARY
  ```

  or

  ```sql
  ORDER BY RANK() OVER (ORDER BY SALARY+BONUS DESC)
  ```

- Rank the departments according to their average total salary.

  ```sql
  SELECT WORKDEPT, AVG(SALARY+BONUS) AS AVG_TOTAL_SALARY,
  RANK() OVER (ORDER BY AVG(SALARY+BONUS) DESC) AS RANK_AVG_SAL
  FROM EMPLOYEE
  GROUP BY WORKDEPT
  ORDER BY RANK_AVG_SAL
  ```

- Rank the employees within a department according to their education level. Having multiple employees with the same rank in the department should not increase the next ranking value.

  ```sql
  SELECT WORKDEPT, EMPNO, LASTNAME, FIRSTNAME, EDLEVEL,
  DENSE_RANK() OVER (PARTITION BY WORKDEPT ORDER BY EDLEVEL DESC) AS RANK_EDLEVEL
  FROM EMPLOYEE
  ORDER BY WORKDEPT, LASTNAME
  ```

- Provide row numbers in the result of a query.

  ```sql
  SELECT ROW_NUMBER() OVER (ORDER BY WORKDEPT, LASTNAME) AS NUMBER,
  LASTNAME, SALARY
  FROM EMPLOYEE
  ORDER BY WORKDEPT, LASTNAME
  ```

- List the top five wage earners.

  ```sql
  SELECT ROW_NUMBER() OVER (ORDER BY SALARY DESC) AS NUMBER,
  EMPNO, LASTNAME, SALARY
  FROM EMPLOYEE
  ORDER BY SALARY DESC
  LIMIT 5
  ```
OLAP specifications

SELECT EMPNO, LASTNAME, FIRSTNME, TOTAL_SALARY, RANK_SALARY
FROM (SELECT EMPNO, LASTNAME, FIRSTNME, SALARY+BONUS AS TOTAL_SALARY,
       RANK() OVER (ORDER BY SALARY+BONUS DESC) AS RANK_SALARY
    FROM EMPLOYEE)
WHERE RANK_SALARY < 6
ORDER BY RANK_SALARY

Note that a nested table expression was used to first compute the result, including the rankings, before the rank could be used in the WHERE clause. A common table expression could also have been used.

- For each department, list employee salaries and show how much less each person makes compared to the employee in that department with the next highest salary.

SELECT EMPNO, WORKDEPT, LASTNAME, FIRSTNME, JOB, SALARY,
       LEAD(SALARY, 1) OVER (PARTITION BY WORKDEPT ORDER BY SALARY) - SALARY AS DELTA_SALARY
FROM EMPLOYEE
ORDER BY WORKDEPT, SALARY

- Calculate an employee’s salary relative to the salary of the employee who was first hired for the same type of job.

SELECT JOB, HIREDATE, EMPNO, LASTNAME, FIRSTNME, SALARY,
       FIRST_VALUE(SALARY) OVER (PARTITION BY JOB ORDER BY HIREDATE) AS FIRST_SALARY,
       SALARY - FIRST_VALUE(SALARY) OVER (PARTITION BY JOB ORDER BY HIREDATE) AS DELTA_SALARY
FROM EMPLOYEE
ORDER BY JOB, HIREDATE

- Calculate the average close price for stock XYZ during the month of January, 2006. If a stock doesn’t trade on a given day, its close price in the DAILYSTOCKDATA table is the null value. Instead of returning the null value for days that a stock doesn’t trade, use the COALESCE function and LAG function to return the close price for the most recent day the stock was traded. Limit the search for a previous not-null close value to one month prior to January 1st, 2006.

WITH V1(SYMBOL, TRADINGDATE, CLOSEPRICE) AS
( SELECT SYMBOL, TRADINGDATE,
       COALESCE(CLOSEPRICE,
       LAG(CLOSEPRICE,
       1,
       CAST(NULL AS DECIMAL(8,2)),
       'IGNORE NULLS')
       OVER (PARTITION BY SYMBOL ORDER BY TRADINGDATE)
    )
FROM DAILYSTOCKDATA
WHERE SYMBOL = 'XYZ' AND TRADINGDATE BETWEEN '2005-12-01' AND '2006-01-31'
)
SELECT SYMBOL, AVG(CLOSEPRICE) AS AVG
FROM V1
WHERE TRADINGDATE BETWEEN '2006-01-01' AND '2006-01-31'
GROUP BY SYMBOL

- Calculate the 30-day moving average for stocks ABC and XYZ during the year 2005.

WITH V1(SYMBOL, TRADINGDATE, MOVINGAVG30DAY) AS
( SELECT SYMBOL, TRADINGDATE,
       AVG(CLOSEPRICE) OVER (PARTITION BY SYMBOL ORDER BY TRADINGDATE ROWS BETWEEN 29 PRECEDING AND CURRENT ROW)
    )
FROM DAILYSTOCKDATA
WHERE SYMBOL IN ('ABC', 'XYZ')
GROUP BY SYMBOL, TRADINGDATE
ORDER BY SYMBOL, TRADINGDATE

Note that a nested table expression was used to first compute the result, including the rankings, before the rank could be used in the WHERE clause. A common table expression could also have been used.
OLAP specifications

FROM DAILYSTOCKDATA
WHERE SYMBOL IN ('ABC', 'XYZ')
    AND TRADINGDATE BETWEEN DATE('2005-01-01') - 2 MONTHS
    AND '2005-12-31'
)
SELECT SYMBOL, TRADINGDATE, MOVINGAVG30DAY
FROM V1
WHERE TRADINGDATE BETWEEN '2005-01-01' AND '2005-12-31'
ORDER BY SYMBOL, TRADINGDATE

- Use an expression to define the cursor position and query a sliding window of 50 rows before that position.

SELECT DATE, FIRST_VALUE(CLOSEPRICE + 100) OVER
    (PARTITION BY SYMBOL
     ORDER BY DATE
     ROWS BETWEEN 50 PRECEDING AND 1 PRECEDING) AS FV
FROM DAILYSTOCKDATA
ORDER BY DATE
ROW CHANGE expression

**ROW CHANGE expression**

row-change-expression:

```
| ROW CHANGE | TOKEN | TIMESTAMP | FOR table-designator |
```

A ROW CHANGE expression returns a token or a timestamp that represents the last change to a row.

**TOKEN**

Specifies that a BIGINT value representing a relative point in the modification sequence of a row is to be returned. If the row has not been changed, the result is a token that represents when the initial value was inserted. The result can be null. ROW CHANGE TOKEN is not deterministic.

**TIMESTAMP**

Specifies that a TIMESTAMP value representing the last time that a row was changed is to be returned. If the row has not been changed, the result is the time that the initial value was inserted. The result can be null. ROW CHANGE TIMESTAMP is not deterministic.

**FOR table-designator**

Identifies the table in which the expression is referenced. The table-designator must uniquely identify a base table, view, or nested table expression (SQLSTATE 42867). If table-designator identifies a view or a nested table expression, the ROW CHANGE expression returns the TOKEN or TIMESTAMP of the base table of the view or nested table expression. The view or nested table expression must contain only one base table in its outer subselect (SQLSTATE 42867). If the table-designator is a view or nested table expression, it must be deletable (SQLSTATE 42703). For information about deletable views, see the “Notes” section of “CREATE VIEW”. The table designator of a ROW CHANGE TIMESTAMP expression must resolve to a base table that contains a row change timestamp column (SQLSTATE 55068).

**Notes**

- The values returned by the ROW CHANGE TOKEN expression can be used with the RID_BIT scalar function by applications that use optimistic locking.

**Examples**

- Return a timestamp value that corresponds to the most recent change to each row from the EMPLOYEE table for employees in department 20. Assume that the EMPLOYEE table has been altered to contain a column defined with the ROW CHANGE TIMESTAMP clause.

  ```sql
  SELECT ROW CHANGE TIMESTAMP FOR EMPLOYEE
  FROM EMPLOYEE WHERE DEPTNO = 20
  ```

- Return a BIGINT value that represents a relative point in the modification sequence of the row corresponding to employee number 3500. Also return the RID_BIT scalar function value that is to be used in an optimistic locking DELETE scenario. Specify the WITH UR option to get the latest ROW CHANGE TOKEN value.

  ```sql
  SELECT ROW CHANGE TOKEN FOR EMPLOYEE, RID_BIT (EMPLOYEE)
  FROM EMPLOYEE WHERE EMPNO = '3500' WITH UR
  ```

The above statement succeeds whether or not there is a row change timestamp column in the EMPLOYEE table. The following searched DELETE statement
ROW CHANGE expression

deletes the row specified by the ROW CHANGE TOKEN and RID_BIT values from the above SELECT statement, assuming the two parameter marker values are set to the values obtained from the above statement.

```
DELETE FROM EMPLOYEE E
    WHERE RID_BIT (E) = ? AND ROW CHANGE TOKEN FOR E = ?
```
Sequence reference

**sequence-reference:**

```
nextval-expression
prevval-expression
```

**nextval-expression:**

```
NEXT VALUE FOR sequence-name
```

**prevval-expression:**

```
PREVIOUS VALUE FOR sequence-name
```

**NEXT VALUE FOR** *sequence-name*

A NEXT VALUE expression generates and returns the next value for the sequence specified by *sequence-name*.

**PREVIOUS VALUE FOR** *sequence-name*

A PREVIOUS VALUE expression returns the most recently generated value for the specified sequence for a previous statement within the current application process. This value can be referenced repeatedly by using PREVIOUS VALUE expressions that specify the name of the sequence. There may be multiple instances of PREVIOUS VALUE expressions specifying the same sequence name within a single statement; they all return the same value. In a partitioned database environment, a PREVIOUS VALUE expression may not return the most recently generated value.

A PREVIOUS VALUE expression can only be used if a NEXT VALUE expression specifying the same sequence name has already been referenced in the current application process, in either the current or a previous transaction (SQLSTATE 51035).

**Notes**

- A new value is generated for a sequence when a NEXT VALUE expression specifies the name of that sequence. However, if there are multiple instances of a NEXT VALUE expression specifying the same sequence name within a query, the counter for the sequence is incremented only once for each row of the result, and all instances of NEXT VALUE return the same value for a row of the result.
- The same sequence number can be used as a unique key value in two separate tables by referencing the sequence number with a NEXT VALUE expression for the first row (this generates the sequence value), and a PREVIOUS VALUE expression for the other rows (the instance of PREVIOUS VALUE refers to the sequence value most recently generated in the current session), as shown below:

  ```sql
  INSERT INTO order(orderno, cutno)
  VALUES (NEXT VALUE FOR order_seq, 123456);
  
  INSERT INTO line_item (orderno, partno, quantity)
  VALUES (PREVIOUS VALUE FOR order_seq, 987654, 1);
  ```
- NEXT VALUE and PREVIOUS VALUE expressions can be specified in the following places:
Sequence reference

- select-statement or SELECT INTO statement (within the select-clause, provided that the statement does not contain a DISTINCT keyword, a GROUP BY clause, an ORDER BY clause, a UNION keyword, an INTERSECT keyword, or EXCEPT keyword)
- INSERT statement (within a VALUES clause)
- INSERT statement (within the select-clause of the fullselect)
- UPDATE statement (within the SET clause (either a searched or a positioned UPDATE statement), except that NEXT VALUE cannot be specified in the select-clause of the fullselect of an expression in the SET clause)
- SET Variable statement (except within the select-clause of the fullselect of an expression; a NEXT VALUE expression can be specified in a trigger, but a PREVIOUS VALUE expression cannot)
- VALUES INTO statement (within the select-clause of the fullselect of an expression)
- CREATE PROCEDURE statement (within the routine-body of an SQL procedure)
- CREATE TRIGGER statement within the triggered-action (a NEXT VALUE expression may be specified, but a PREVIOUS VALUE expression cannot)

- NEXT VALUE and PREVIOUS VALUE expressions cannot be specified (SQLSTATE 428F9) in the following places:
  - Join condition of a full outer join
  - DEFAULT value for a column in a CREATE or ALTER TABLE statement
  - Generated column definition in a CREATE OR ALTER TABLE statement
  - Summary table definition in a CREATE TABLE or ALTER TABLE statement
  - Condition of a CHECK constraint
  - CREATE TRIGGER statement (a NEXT VALUE expression may be specified, but a PREVIOUS VALUE expression cannot)
  - CREATE VIEW statement
  - CREATE METHOD statement
  - CREATE FUNCTION statement
  - An argument list of an XMLQUERY, XMLEXISTS, or XMLTABLE expression

- In addition, a NEXT VALUE expression cannot be specified (SQLSTATE 428F9) in the following places:
  - CASE expression
  - Parameter list of an aggregate function
  - Subquery in a context other than those explicitly allowed above
  - SELECT statement for which the outer SELECT contains a DISTINCT operator
  - Join condition of a join
  - SELECT statement for which the outer SELECT contains a GROUP BY clause
  - SELECT statement for which the outer SELECT is combined with another SELECT statement using the UNION, INTERSECT, or EXCEPT set operator
  - Nested table expression
  - Parameter list of a table function
  - WHERE clause of the outer-most SELECT statement, or a DELETE or UPDATE statement
  - ORDER BY clause of the outer-most SELECT statement
Sequence reference

- select-clause of the fullselect of an expression, in the SET clause of an
  UPDATE statement
- IF, WHILE, DO ... UNTIL, or CASE statement in an SQL routine

When a value is generated for a sequence, that value is consumed, and the next
time that a value is requested, a new value will be generated. This is true even
when the statement containing the NEXT VALUE expression fails or is rolled
back.

If an INSERT statement includes a NEXT VALUE expression in the VALUES list
for the column, and if an error occurs at some point during the execution of the
INSERT (it could be a problem in generating the next sequence value, or a
problem with the value for another column), then an insertion failure occurs
(SQLSTATE 23505), and the value generated for the sequence is considered to be
consumed. In some cases, reissuing the same INSERT statement might lead to
success.

For example, consider an error that is the result of the existence of a unique
index for the column for which NEXT VALUE was used and the sequence value
generated already exists in the index. It is possible that the next value generated
for the sequence is a value that does not exist in the index and so the
subsequent INSERT would succeed.

- **Scope of PREVIOUS VALUE:** The value of PREVIOUS VALUE persists until the
  next value is generated for the sequence in the current session, the sequence is
dropped or altered, or the application session ends. The value is unaffected by
COMMIT or ROLLBACK statements. The value of PREVIOUS VALUE cannot be
directly set and is a result of executing the NEXT VALUE expression for the
sequence.

  A technique commonly used, especially for performance, is for an application or
product to manage a set of connections and route transactions to an arbitrary
connection. In these situations, the availability of the PREVIOUS VALUE for a
sequence should be relied on only until the end of the transaction. Examples of
where this type of situation can occur include applications that use XA
protocols, use connection pooling, use the connection concentrator, and use
HADR to achieve failover.

- If in generating a value for a sequence, the maximum value for the sequence is
  exceeded (or the minimum value for a descending sequence) and cycles are not
  permitted, then an error occurs (SQLSTATE 23522). In this case, the user could
ALTER the sequence to extend the range of acceptable values, or enable cycles
for the sequence, or DROP and CREATE a new sequence with a different data
type that has a larger range of values.

  For example, a sequence may have been defined with a data type of SMALLINT,
and eventually the sequence runs out of assignable values. DROP and re-create
the sequence with the new definition to redefine the sequence as INTEGER.

- A reference to a NEXT VALUE expression in the select statement of a cursor
  refers to a value that is generated for a row of the result table. A sequence value
is generated for a NEXT VALUE expression for each row that is fetched from the
database. If blocking is done at the client, the values may have been generated
at the server prior to the processing of the FETCH statement. This can occur
when there is blocking of the rows of the result table. If the client application
does not explicitly FETCH all the rows that the database has materialized, then
the application will not see the results of all the generated sequence values (for
the materialized rows that were not returned).

- A reference to a PREVIOUS VALUE expression in the select statement of a
cursor refers to a value that was generated for the specified sequence prior to
the opening of the cursor. However, closing the cursor can affect the values
Sequence reference

returned by PREVIOUS VALUE for the specified sequence in subsequent statements, or even for the same statement in the event that the cursor is reopened. This would be the case when the select statement of the cursor included a reference to NEXT VALUE for the same sequence name.

- **Compatibilities**
  - For compatibility with previous versions of DB2:
    - NEXTVAL and PREVVAL can be specified in place of NEXT VALUE and PREVIOUS VALUE
  - For compatibility with IBM IDS:
    - `sequence-name.NEXTVAL` can be specified in place of `NEXT VALUE FOR sequence-name`
    - `sequence-name.CURRVAL` can be specified in place of `PREVIOUS VALUE FOR sequence-name`

**Examples**

Assume that there is a table called "order", and that a sequence called "order_seq" is created as follows:

```
CREATE SEQUENCE order_seq
    START WITH 1
    INCREMENT BY 1
    NO MAXVALUE
    NO CYCLE
    CACHE 24
```

Following are some examples of how to generate an "order_seq" sequence number with a NEXT VALUE expression:

```
INSERT INTO order(orderno, custno)
    VALUES (NEXT VALUE FOR order_seq, 123456);
```

or

```
UPDATE order
    SET orderno = NEXT VALUE FOR order_seq
    WHERE custno = 123456;
```

or

```
VALUES NEXT VALUE FOR order_seq INTO :hv_seq;
```
Subtype treatment

Subtype treatment

subtype-treatment:

[TREAT(expression AS data-type)]

The subtype-treatment is used to cast a structured type expression into one of its subtypes. The static type of expression must be a user-defined structured type, and that type must be the same type as, or a supertype of, data-type. If the type name in data-type is unqualified, the SQL path is used to resolve the type reference. The static type of the result of subtype-treatment is data-type, and the value of the subtype-treatment is the value of the expression. At run time, if the dynamic type of the expression is not data-type or a subtype of data-type, an error is returned (SQLSTATE 0D000).

Example

- If an application knows that all column object instances in a column CIRCLE_COL have the dynamic type COLOREDCIRCLE, use the following query to invoke the method RGB on such objects. Assume the existence of a table called RINGS, with a column CIRCLE_COL of structured type CIRCLE. Also, assume that COLOREDCIRCLE is a subtype of CIRCLE and that the method RGB has been defined previously for COLOREDCIRCLE as RGB()

  RETURNS DOUBLE.

  SELECT TREAT (CIRCLE_COL AS COLOREDCIRCLE)..RGB()
  FROM RINGS

  At run time, if there are instances of dynamic type CIRCLE, an error is raised (SQLSTATE 0D000). This error can be avoided by using the TYPE predicate in a CASE expression, as follows:

  SELECT (CASE
    WHEN CIRCLE_COL IS OF (COLOREDCIRCLE)
      THEN TREAT (CIRCLE_COL AS COLOREDCIRCLE)..RGB()
    ELSE NULL
    END)
  FROM RINGS
Determining data types of untyped expressions

An untyped expression refers to the usage of a parameter marker or null value which is specified without a target data type associated with it.

Untyped expressions can be used in SQL statements as long as one of the following conditions is true:

- A PREPARE statement is being executed to compile the SQL statement; the client interface is using deferred prepare; and the registry variable, DB2_DEFERRED_PREPARE_SEMANTICS is set to YES. In this case, any untyped parameter marker derives its data type based on the input descriptor associated with the subsequent OPEN or EXECUTE statement. The length attribute is set to the maximum of the length according to the UNTYPED row, as described in the Table 21 on page 199 in “Functions” and the length as determined from the tables below. For data types not listed as a target type in Table 21 on page 199 in “Functions”, the length from the input descriptor associated with the subsequent OPEN or EXECUTE statement will be used. The data types and lengths may be modified depending on the usage of the untyped parameter marker in the SQL statement.

- The data type can be determined based on the context in the SQL statement. These locations and the resulting data types are shown in the following table. The locations are grouped into expressions, predicates, built-in functions, and user-defined routines to assist in determining the applicability of an untyped expression. If the data type cannot be determined based on the context, an error is issued.

For some cases not listed, untyped expressions in a select list will be resolved to a data type determined based on the usage in the SQL statement.

The code page of the untyped expression is determined by the context. Where there is no context, the code page is the same as if the untyped expression was cast to a VARCHAR data type.

Table 24. Untyped Expression Usage in Expressions (Including Select List, CASE, and VALUES)

<table>
<thead>
<tr>
<th>Untyped Expression Location</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alone in a select list</td>
<td>Error if the untyped expression is unnamed or is named but not subsequently referenced in the SQL statement. If the untyped expression is named and subsequently referenced in the SQL statement, then the data type may be determined from the subsequent usage. For more information, refer to the &quot;Determining data type from usage&quot; note that follows this table.</td>
</tr>
<tr>
<td>Both operands of a single arithmetic operator, after considering operator precedence and order of operation rules</td>
<td>DECFLOAT(34)</td>
</tr>
</tbody>
</table>

Includes cases such as:

\[(? + ?) + 10\]
### Determining data types of untyped expressions

Table 24. Untyped Expression Usage in Expressions (Including Select List, CASE, and VALUES) (continued)

<table>
<thead>
<tr>
<th>Untyped Expression Location</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>One operand of a single operator in an arithmetic expression (not a datetime expression)</td>
<td>The data type of the other operand</td>
</tr>
<tr>
<td>Includes cases such as: ? + (? * 10)</td>
<td>DECIMAL(15,0)</td>
</tr>
<tr>
<td>Labelled duration within a datetime expression (note that the portion of a labelled duration that indicates the type of units cannot be a parameter marker)</td>
<td>Error</td>
</tr>
<tr>
<td>Any other operand of a datetime expression (for example, ‘timecol + ?’ or ‘? - datecol’)</td>
<td>Error</td>
</tr>
<tr>
<td>Both operands of a CONCAT operator</td>
<td>VARCHAR(254)</td>
</tr>
<tr>
<td>One operand of a CONCAT operator when the other operand is a non-CLOB character data type</td>
<td>If one operand is either CHAR(n) or VARCHAR(n), where n is less than 128, the other is VARCHAR(254 - n); in all other cases, the data type is VARCHAR(254)</td>
</tr>
<tr>
<td>One operand of a CONCAT operator, when the other operand is a non-DBCLOB graphic data type</td>
<td>If one operand is either GRAPHIC(n) or VARGRAPHIC(n), where n is less than 64, the other is VARGRAPHIC(127 - n); in all other cases, the data type is VARGRAPHIC(127)</td>
</tr>
<tr>
<td>One operand of a CONCAT operator, when the other operand is a large object string</td>
<td>Same as that of the other operand</td>
</tr>
<tr>
<td>The expression following the CASE keyword in a simple CASE expression</td>
<td>Result of applying the “Rules for result data types” to the expressions following the WHEN keyword that are other than untyped expressions</td>
</tr>
<tr>
<td>At least one of the result-expressions in a CASE expression (both simple and searched), with the rest of the result-expressions being untyped expressions</td>
<td>Error</td>
</tr>
<tr>
<td>Any or all expressions following the WHEN keyword in a simple CASE expression</td>
<td>Result of applying the “Rules for result data types” to the expression following CASE and the expressions following WHEN keyword that are other than an untyped expression</td>
</tr>
<tr>
<td>A result-expression in a CASE expression (both simple and searched), when at least one result-expression is not an untyped expression</td>
<td>Result of applying the “Rules for result data types” to all result-expressions that are other than an untyped expression</td>
</tr>
<tr>
<td>Alone as a column-expression in a single-row VALUES clause that is not within an INSERT statement and not within the VALUES clause of in insert operation of a MERGE statement</td>
<td>Error if the untyped expression is unnamed or is named but not subsequently referenced in the SQL statement. If the untyped expression is named and subsequently referenced in the SQL statement, then the data type may be determined from the subsequent usage. For more information, refer to the “Determining data type from usage” note that follows this table.</td>
</tr>
</tbody>
</table>
### Determining data types of untyped expressions

<table>
<thead>
<tr>
<th>Untyped Expression Location</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alone as a column-expression in a multi-row VALUES clause that is not within an INSERT statement, and for which the column-expressions in the same position in all other row-expressions are untyped expressions</td>
<td>Error if the untyped expression is unnamed or is named but not subsequently referenced in the SQL statement. If the untyped expression is named and subsequently referenced in the SQL statement, then the data type may be determined from the subsequent usage. For more information, refer to the &quot;Determining data type from usage&quot; note that follows this table.</td>
</tr>
<tr>
<td>Alone as a column-expression in a multi-row VALUES clause that is not within an INSERT statement, and for which the expression in the same position of at least one other row-expression is not an untyped expression</td>
<td>Result of applying the “Rules for result data types” on all operands that are other than untyped expressions</td>
</tr>
<tr>
<td>Alone as a column-expression in a single-row VALUES clause within an INSERT statement</td>
<td>The data type of the column. If the column is defined as a user-defined distinct type, it is the source data type of the user-defined distinct type. If the column is defined as a user-defined structured type, it is the structured type, also indicating the return type of the transform function.</td>
</tr>
<tr>
<td>Alone as a column-expression in a multi-row VALUES clause within an INSERT statement</td>
<td>The data type of the column. If the column is defined as a user-defined distinct type, it is the source data type of the user-defined distinct type. If the column is defined as a user-defined structured type, it is the structured type, also indicating the return type of the transform function.</td>
</tr>
<tr>
<td>Alone as a column-expression on the right side of assignment-clause for an update operation of a MERGE statement</td>
<td>Error if the untyped expression is unnamed or is named but not subsequently referenced in the SQL statement. If the untyped expression is named and subsequently referenced in the SQL statement, then the data type may be determined from the subsequent usage. For more information, refer to the &quot;Determining data type from usage&quot; note that follows this table.</td>
</tr>
<tr>
<td>Alone as a column-expression in the VALUES clause of an insert operation of a MERGE statement</td>
<td>The data type of the column. If the column is defined as a user-defined distinct type, it is the source data type of the user-defined distinct type. If the column is defined as a user-defined structured type, it is the structured type, also indicating the return type of the transform function.</td>
</tr>
<tr>
<td>Alone as a column-expression on the right side of assignment-clause for an update operation of a MERGE statement</td>
<td>The data type of the column. If the column is defined as a user-defined distinct type, it is the source data type of the user-defined distinct type. If the column is defined as a user-defined structured type, it is the structured type, also indicating the return type of the transform function.</td>
</tr>
</tbody>
</table>
### Determining data types of untyped expressions

#### Table 24. Untyped Expression Usage in Expressions (Including Select List, CASE, and VALUES) (continued)

<table>
<thead>
<tr>
<th>Untyped Expression Location</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alone as a column-expression on the right side of a SET clause in an UPDATE statement</td>
<td>The data type of the column. If the column is defined as a user-defined distinct type, it is the source data type of the user-defined distinct type. If the column is defined as a user-defined structured type, it is the structured type, also indicating the return type of the transform function.</td>
</tr>
<tr>
<td>As a value on the right side of a SET special register statement</td>
<td>The data type of the special register</td>
</tr>
<tr>
<td>Argument of the TABLESAMPLE clause of the tablesample-clause of a table-reference</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>Argument of the REPEATABLE subclause of the tablesample-clause of a table-reference</td>
<td>INTEGER</td>
</tr>
<tr>
<td>As a value in a FREE LOCATOR statement</td>
<td>Locator</td>
</tr>
<tr>
<td>As a value for the password in a SET ENCRYPTION PASSWORD statement</td>
<td>VARCHAR(128)</td>
</tr>
</tbody>
</table>

**Note:**

**Determining data type from usage**

The following is an example of how the data type for an untyped expression can be determined from subsequent usage:

If the named untyped expression is subsequently referenced in a comparison operator, it will then have the data type of the other operand. If there are multiple references of the named untyped expression in the SQL statement, the data type, length, precision, scale, and code page that is independently determined for each of those references must be identical or an error is returned.

#### Table 25. Untyped Expression Usage in Predicates

<table>
<thead>
<tr>
<th>Untyped Expression Location</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Both operands of a comparison operator</td>
<td>VARCHAR(254)</td>
</tr>
<tr>
<td>One operand of a comparison operator, when the other operand is other than an untyped expression</td>
<td>The data type of the other operand</td>
</tr>
<tr>
<td>All operands of a BETWEEN predicate</td>
<td>VARCHAR(254)</td>
</tr>
<tr>
<td>Two operands of a BETWEEN predicate</td>
<td>Same as that of the only typed expression</td>
</tr>
<tr>
<td>Only one operand of a BETWEEN predicate</td>
<td>Result of applying the “Rules for result data types” on all operands that are other than untyped expressions</td>
</tr>
<tr>
<td>All operands of an IN predicate, for example, ? IN (?,?,?)</td>
<td>VARCHAR(254)</td>
</tr>
<tr>
<td>The first operand of an IN predicate, when the right hand side is a fullselect, for example, IN (fullselect)</td>
<td>Data type of the selected column</td>
</tr>
</tbody>
</table>
### Determining data types of untyped expressions

**Table 25. Untyped Expression Usage in Predicates (continued)**

<table>
<thead>
<tr>
<th>Untyped Expression Location</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>The first operand of an IN predicate, when the right hand side is not a subselect, for example, ? IN (?,A,B), or ? IN (A,?,B,?)</td>
<td>Result of applying the &quot;Rules for result data types&quot; on all operands of the IN list (operands to the right of the IN keyword) that are other than untyped expressions</td>
</tr>
<tr>
<td>Any or all operands of the IN list of the IN predicate, for example, A IN (?,B, ?)</td>
<td>Result of applying the &quot;Rules for result data types&quot; on all operands of the IN predicate (operands to the left and right of the IN keyword) that are other than untyped expressions</td>
</tr>
<tr>
<td>Both the operand in a row-value-expression of an IN predicate, and the corresponding result column of the fullselect, for example, (c1, ?) IN (SELECT c1, ? FROM ...)</td>
<td>VARCHAR(254)</td>
</tr>
<tr>
<td>Any operands in a row-value-expression of an IN predicate, for example, (c1,?) IN fullselect</td>
<td>Data type of the corresponding result column of the fullselect</td>
</tr>
<tr>
<td>Any select list items in a subquery if a row-value-expression is specified in an IN predicate, for example, (c1,c2) IN (SELECT?, c1, FROM ...)</td>
<td>Data type of the corresponding operand in the row-value-expression</td>
</tr>
<tr>
<td>All three operands of the LIKE predicate</td>
<td>Match expression (operand 1) and pattern expression (operand 2) are VARCHAR(32672); escape expression (operand 3) is VARCHAR(2)</td>
</tr>
<tr>
<td>The match expression of the LIKE predicate when either the pattern expression or the escape expression is other than an untyped expression</td>
<td>Either VARCHAR(32672) or VARGRAPHIC(16336), depending on the data type of the first operand that is not an untyped expression</td>
</tr>
<tr>
<td>The pattern expression of the LIKE predicate when either the match expression or the escape expression is other than an untyped expression</td>
<td>Either VARCHAR(32672) or VARGRAPHIC(16336), depending on the data type of the first operand that is not an untyped expression; if the data type of the match expression is BLOB, the data type of the pattern expression is assumed to be BLOB(32672)</td>
</tr>
<tr>
<td>The escape expression of the LIKE predicate when either the match expression or the pattern expression is other than an untyped expression</td>
<td>Either VARCHAR(2) or VARGRAPHIC(1), depending on the data type of the first operand that is not an untyped expression; if the data type of the match expression or pattern expression is BLOB, the data type of the escape expression is assumed to be BLOB(1)</td>
</tr>
<tr>
<td>Operand of the NULL predicate</td>
<td>VARCHAR(254)</td>
</tr>
</tbody>
</table>

**Table 26. Untyped Expression Usage in Built-in Functions**

<table>
<thead>
<tr>
<th>Untyped Parameter Marker Location</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>All arguments of COALESCE, MIN, MAX, NULLIF, or VALUE</td>
<td>Error</td>
</tr>
</tbody>
</table>
Table 26. Untyped Expression Usage in Built-in Functions (continued)

<table>
<thead>
<tr>
<th>Untyped Parameter Marker Location</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any argument of COALESCE, MIN, MAX, NULLIF, or VALUE, when at least one argument is other than an untyped parameter marker</td>
<td>Result of applying the “Rules for result data types” on all arguments that are other than untyped parameter markers</td>
</tr>
<tr>
<td>First argument of DAYNAME</td>
<td>TIMESTAMP(12)</td>
</tr>
<tr>
<td>The argument of DIGITS</td>
<td>DECIMAL(31,6)</td>
</tr>
<tr>
<td>First argument of MONTHNAME</td>
<td>TIMESTAMP(12)</td>
</tr>
<tr>
<td>POSSTR (both arguments)</td>
<td>Both arguments are VARCHAR(32672)</td>
</tr>
<tr>
<td>POSSTR (one argument, when the other argument is a character data type)</td>
<td>VARCHAR(32672)</td>
</tr>
<tr>
<td>POSSTR (one argument, when the other argument is a graphic data type)</td>
<td>VARGRAPHIC(16336)</td>
</tr>
<tr>
<td>POSSTR (the search-string argument, when the other argument is a BLOB)</td>
<td>BLOB(32672)</td>
</tr>
<tr>
<td>First argument of SUBSTR</td>
<td>VARCHAR(32672)</td>
</tr>
<tr>
<td>Second and third argument of SUBSTR</td>
<td>INTEGER</td>
</tr>
<tr>
<td>Second and third arguments of TRANSLATE</td>
<td>VARCHAR(32672) if the first argument is a character type; VARGRAPHIC(16336) if the first argument is a graphic type</td>
</tr>
<tr>
<td>Fourth argument of TRANSLATE</td>
<td>VARCHAR(1) if the first argument is a character type; VARGRAPHIC(1) if the first argument is a graphic type</td>
</tr>
<tr>
<td>The Second argument of TIMESTAMP</td>
<td>TIME</td>
</tr>
<tr>
<td>First argument of VARCHAR_FORMAT</td>
<td>TIMESTAMP(12)</td>
</tr>
<tr>
<td>First argument of TIMESTAMP_FORMAT</td>
<td>VARCHAR(254)</td>
</tr>
<tr>
<td>First argument of XMLVALIDATE</td>
<td>XML</td>
</tr>
<tr>
<td>First argument of XMLCOMMENT</td>
<td>VARCHAR(32672)</td>
</tr>
<tr>
<td>First argument of XMLTEXT</td>
<td>VARCHAR(32672)</td>
</tr>
<tr>
<td>Second argument of XMLPI</td>
<td>VARCHAR(32672)</td>
</tr>
<tr>
<td>First argument of XMLSERIALIZE</td>
<td>XML</td>
</tr>
<tr>
<td>First argument of XMLDOCUMENT</td>
<td>XML</td>
</tr>
<tr>
<td>First argument of XMLXSROBJECTID</td>
<td>XML</td>
</tr>
<tr>
<td>All arguments of XMLCONCAT</td>
<td>XML</td>
</tr>
<tr>
<td>Second argument of TRIM_ARRAY</td>
<td>BIGINT</td>
</tr>
<tr>
<td>Array index of an ARRAY</td>
<td>BIGINT</td>
</tr>
<tr>
<td>Unary minus</td>
<td>DECFLOAT(34)</td>
</tr>
<tr>
<td>Unary plus</td>
<td>DECFLOAT(34)</td>
</tr>
<tr>
<td>All other arguments of all other scalar functions</td>
<td>The data type of the parameter of the function definition as determined by function resolution. The length of the argument is derived based on Table 21 on page 199 in Function Resolution section.</td>
</tr>
<tr>
<td>Arguments of a aggregate function</td>
<td>Error</td>
</tr>
</tbody>
</table>
## Determining data types of untyped expressions

<table>
<thead>
<tr>
<th>Untyped Parameter Marker Location</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Argument of a function</td>
<td>The data type and length of the parameter, as defined when the function was created.</td>
</tr>
<tr>
<td>Argument of a method</td>
<td>Error</td>
</tr>
<tr>
<td>Argument of a procedure</td>
<td>The data type of the parameter, as defined when the procedure was created</td>
</tr>
</tbody>
</table>
Row expression

A row expression specifies a row of data that could have a specific user-defined row type or the built-in data type ROW.

Authorization

The use of some of the row expressions may require having the appropriate authorization. For these row expressions, the privileges held by the authorization ID of the statement must include the following authorization:

- **row-variable.** For information about authorization considerations when *row-variable* is a global variable, see “Global variables”.
- **row-function-invocation.** The authorization to execute the function. For information about authorization considerations, see “Function invocation”.
- **expression.** Authorizations might be required for the use of certain expressions referenced in a *row-expression*. For information about authorization considerations, see “Expressions”.

Syntax

```sql
row-expression:
  row-variable
  row-case-expression
  row-cast-specification
  row-array-element-specification
  row-function-invocation
```

Description

- **row-variable**
  A variable that is defined with row type.

- **row-case-expression**
  A case-expression that returns a row type.

- **row-cast-specification**
  A CAST that returns a row type.

- **row-array-element-specification**
  An array-element-specification of an array with row type elements.

- **row-function-invocation**
  A function-invocation of a user-defined function that has a return type that is a row type. The function could return a user-defined row type or the data type ROW with defined field names and field types.

Notes

- Row expressions can be used to generate a row within SQL PL contexts.

Example
A *predicate* specifies a condition that is true, false, or unknown about a given value, row, or group.

The following rules apply to all types of predicates:

- All values specified in a predicate must be compatible.
- An expression used in a basic, quantified, IN, or BETWEEN predicate must not result in a character string with a length attribute greater than 4000, a graphic string with a length attribute greater than 2000, or a LOB string of any size.
- The value of a host variable can be null (that is, the variable may have a negative indicator variable).
- The code page conversion of operands of predicates involving two or more operands, with the exception of LIKE, is done according to the rules for string conversions.
- Use of a structured type value is limited to the NULL predicate and the TYPE predicate.
- In a Unicode database, all predicates that accept a character or graphic string will accept any string type for which conversion is supported.

A fullselect is a form of the SELECT statement that, when used in a predicate, is also called a *subquery*. 
Predicate processing for queries

A predicate is an element of a search condition that expresses or implies a comparison operation. Predicates can be grouped into four categories that are determined by how and when the predicate is used in the evaluation process. The categories are listed below, ordered in terms of performance starting with the most favorable:

- **Range delimiting predicates** are those used to bracket an index scan; they provide start or stop key values for the index search. These predicates are evaluated by the index manager.
- **Index sargable predicates** are not used to bracket a search, but are evaluated from the index if one is chosen, because the columns involved in the predicate are part of the index key. These predicates are also evaluated by the index manager.
- **Data sargable predicates** are predicates that cannot be evaluated by the index manager, but can be evaluated by Data Management Services (DMS). Typically, these predicates require the access of individual rows from a base table. If necessary, DMS will retrieve the columns needed to evaluate the predicate, as well as any others to satisfy the columns in the SELECT list that could not be obtained from the index.
- **Residual predicates** are those that require I/O beyond the simple accessing of a base table. Examples of residual predicates include those using quantified subqueries (subqueries with ANY, ALL, SOME, or IN), or reading large object (LOB) data that is stored separately from the table. These predicates are evaluated by Relational Data Services (RDS) and are the most expensive of the four categories of predicates.

The following table provides examples of various predicates and identifies their type based on the context in which they are used.

**Note:** In these examples, assume that a multi-column ascending index exists on (c1, c2, c3) and is used in evaluating the predicates where appropriate. If any column in the index is in descending order, the start and stop keys might be switched for range delimiting predicates.

<table>
<thead>
<tr>
<th>Predicates</th>
<th>Column c1</th>
<th>Column c2</th>
<th>Column c3</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1 = 1 and c2 = 2 and c3 = 3</td>
<td>Range delimiting (start-stop)</td>
<td>Range delimiting (start-stop)</td>
<td>Range delimiting (start-stop)</td>
<td>The equality predicates on all the columns of the index can be applied as start-stop keys.</td>
</tr>
<tr>
<td>c1 = 1 and c2 = 2 and c3 ≥ 3</td>
<td>Range delimiting (start-stop)</td>
<td>Range delimiting (start-stop)</td>
<td>Range delimiting (start)</td>
<td>Columns c1 and c2 are bound by equality predicates, and the predicate on c3 is only applied as a start key.</td>
</tr>
<tr>
<td>c1 ≥ 1 and c2 = 2</td>
<td>Range delimiting (start)</td>
<td>Range delimiting (start-stop)</td>
<td>Not applicable</td>
<td>The leading column c1 has a ≥ predicate and can be used as a start key. The following column c2 has an equality predicate, and therefore can also be applied as a start-stop key.</td>
</tr>
</tbody>
</table>
## Predicate processing for queries

**Table 28. Predicate processing for different queries (continued)**

<table>
<thead>
<tr>
<th>Predicates</th>
<th>Column c1</th>
<th>Column c2</th>
<th>Column c3</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>c1 = 1 and c3 = 3</code></td>
<td>Range delimiting (start-stop)</td>
<td>Not applicable</td>
<td>Index sargable</td>
<td>The predicate on c3 cannot be used as a start-stop key, because there is no predicate on c2. It can, however, be applied as an index sargable predicate.</td>
</tr>
<tr>
<td><code>c1 = 1 and c2 &gt; 2 and c3 = 3</code></td>
<td>Range delimiting (start-stop)</td>
<td>Range delimiting (start)</td>
<td>Index sargable</td>
<td>The predicate on c3 cannot be applied as a start-stop predicate because the previous column has a &gt; predicate. Had it been a ≥ instead, we would be able to use it as a start-stop key.</td>
</tr>
<tr>
<td><code>c1 = 1 and c2 ≤ 2 and c4 = 4</code></td>
<td>Range delimiting (start-stop)</td>
<td>Range delimiting (stop)</td>
<td>Data sargable</td>
<td>Here the predicate on c2 is a ≤ predicate. It can be used as a stop key. The predicate on c4 cannot be applied on the index and is applied as a data sargable predicate during the FETCH.</td>
</tr>
<tr>
<td><code>c2 = 2 and UDF_with_external_action(c4)</code></td>
<td>Not applicable</td>
<td>Index sargable</td>
<td>Residual</td>
<td>The leading column c1 does not have a predicate, and therefore the predicate on c2 can be applied as an index sargable predicate where the whole index is scanned. The predicate involving the user-defined function with external action is applied as a residual predicate.</td>
</tr>
<tr>
<td><code>c1 = 1 or c2 = 2</code></td>
<td>Index sargable</td>
<td>Index sargable</td>
<td>Not applicable</td>
<td>The presence of an OR does not allow us this multi-column index to be used as start-stop keys. This might have been possible had there been two indexes, one with a leading column on c1, and the other with a leading column on c2, and the DB2 optimizer chose an “index-ORing” plan. However, in this case the two predicates are treated as index sargable predicates.</td>
</tr>
<tr>
<td><code>c1 &lt; 5 and (c2 = 2 or c3 = 3)</code></td>
<td>Range delimiting (stop)</td>
<td>Index sargable</td>
<td>Index sargable</td>
<td>Here the leading column c1 is exploited to stop the index scan from using the predicate with a stop key. The OR predicate on c2 and c3 are applied as index sargable predicates.</td>
</tr>
</tbody>
</table>

The DB2 optimizer employs the query rewrite mechanism to transform many complex user-written predicates into better performing queries, as shown in the
### Predicate processing for queries

The following table:

<table>
<thead>
<tr>
<th>Original predicate or query</th>
<th>Optimized predicates</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1 between 5 and 10</td>
<td>c1 ( \geq 5 ) and c1 ( \leq 10 )</td>
<td>The BETWEEN predicates are rewritten into the equivalent range delimiting predicates so that they can be used internally as though the user specified the range delimiting predicates.</td>
</tr>
<tr>
<td>c1 not between 5 and 10</td>
<td>c1 &lt; 5 or c1 &gt; 10</td>
<td>The presence of the OR predicate does not allow the use of a start-stop key unless the DB2 optimizer chooses an index-ORing plan.</td>
</tr>
<tr>
<td>SELECT * FROM t1 WHERE EXISTS (SELECT c1 FROM t2 WHERE t1.c1 = t2.c1)</td>
<td>SELECT t1.* FROM t1 EJOIN t2 WHERE t1.c1 = t2.c1</td>
<td>The subquery might be transformed into a join.</td>
</tr>
<tr>
<td>SELECT * FROM t1 WHERE t1.c1 IN (SELECT c1 FROM t2)</td>
<td>SELECT t1.* FROM t1 EJOIN t2 WHERE t1.c1 = t2.c1</td>
<td>This is similar to the transformation for the EXISTS predicate example above.</td>
</tr>
<tr>
<td>c1 like ‘abc%’</td>
<td>c1 ( \geq ) ‘abc X X X’ and c1 ( \leq ) ‘abc Y Y Y’</td>
<td>If we have c1 as the leading column of an index, DB2 generates these predicates so that they can be applied as range-delimiting start-stop predicates. Here the characters X and Y are symbolic of the lowest and highest collating character.</td>
</tr>
<tr>
<td>c1 like ‘abc%def’</td>
<td>c1 ( \geq ) ‘abc X X X’ and c1 ( \leq ) ‘abc Y Y Y’ and c1 like ‘abc%def’</td>
<td>This is similar to the previous case, except that we have to also apply the original predicate as an index sargable predicate. This ensures that the characters def match correctly.</td>
</tr>
</tbody>
</table>
Search conditions

A search condition specifies a condition that is “true,” “false,” or “unknown” about a given value, row, or group.

The result of a search condition is derived by application of the specified logical operators (AND, OR, NOT) to the result of each specified predicate. If logical operators are not specified, the result of the search condition is the result of the specified predicate.

AND and OR are defined in Table 30 in which P and Q are any predicates:

<table>
<thead>
<tr>
<th>P</th>
<th>Q</th>
<th>P AND Q</th>
<th>P OR Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>Unknown</td>
<td>Unknown</td>
<td>True</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>Unknown</td>
<td>False</td>
<td>Unknown</td>
</tr>
<tr>
<td>Unknown</td>
<td>True</td>
<td>Unknown</td>
<td>True</td>
</tr>
<tr>
<td>Unknown</td>
<td>False</td>
<td>False</td>
<td>Unknown</td>
</tr>
<tr>
<td>Unknown</td>
<td>Unknown</td>
<td>Unknown</td>
<td>Unknown</td>
</tr>
</tbody>
</table>

NOT(true) is false, NOT(false) is true, and NOT(unknown) is unknown.

Search conditions within parentheses are evaluated first. If the order of evaluation is not specified by parentheses, NOT is applied before AND, and AND is applied before OR. The order in which operators at the same precedence level are evaluated is undefined to allow for optimization of search conditions.
**SELECTIVITY** value

The SELECTIVITY clause is used to indicate to DB2 what the expected selectivity percentage is for the predicate. SELECTIVITY can be specified only when the predicate is a user-defined predicate.

A user-defined predicate is a predicate that consists of a user-defined function invocation, in the context of a predicate specification that matches the predicate specification on the PREDICATES clause of CREATE FUNCTION. For example, if the function `foo` is defined with PREDICATES WHEN=1..., then the following use of SELECTIVITY is valid:

```sql
SELECT * FROM STORES WHERE foo(parm,parm) = 1 SELECTIVITY 0.004
```

The selectivity value must be a numeric literal value in the inclusive range from 0 to 1 (SQLSTATE 42615). If SELECTIVITY is not specified, the default value is 0.01 (that is, the user-defined predicate is expected to filter out all but one percent of all the rows in the table). The SELECTIVITY default can be changed for any given function by updating its SELECTIVITY column in the SYSSTAT.ROUTINES view. An error will be returned if the SELECTIVITY clause is specified for a non user-defined predicate (SQLSTATE 428E5).

A user-defined function (UDF) can be applied as a user-defined predicate and, hence, is potentially applicable for index exploitation if:

- the predicate specification is present in the CREATE FUNCTION statement
- the UDF is invoked in a WHERE clause being compared (syntactically) in the same way as specified in the predicate specification
- there is no negation (NOT operator)

**Examples**

In the following query, the `within` UDF specification in the WHERE clause satisfies all three conditions and is considered a user-defined predicate.

```sql
SELECT * FROM customers WHERE within(location, :sanJose) = 1 SELECTIVITY 0.2
```

However, the presence of `within` in the following query is not index-exploitable due to negation and is not considered a user-defined predicate.

```sql
SELECT * FROM customers WHERE NOT(within(location, :sanJose) = 1) SELECTIVITY 0.3
```
Search conditions

In the next example, consider identifying customers and stores that are within a certain distance of each other. The distance from one store to another is computed by the radius of the city in which the customers live.

```sql
SELECT *
FROM customers, stores
WHERE distance(customers.loc, stores.loc) < CityRadius(stores.loc) SELECTIVITY 0.02
```

In the above query, the predicate in the WHERE clause is considered a user-defined predicate. The result produced by CityRadius is used as a search argument to the range producer function.

However, since the result produced by CityRadius is used as a range producer function, the above user-defined predicate will not be able to make use of the index extension defined on the stores.loc column. Therefore, the UDF will make use of only the index defined on the customers.loc column.
Basic predicate

A basic predicate compares two values.

If the value of either operand is null, the result of the predicate is unknown. Otherwise the result is either true or false.

For values \( x \) and \( y \):

**Predicate**

Is True If and Only If...

\[
\begin{align*}
& x = y & \text{\( x \) is equal to \( y \)} \\
& x \neq y & \text{\( x \) is not equal to \( y \)} \\
& x < y & \text{\( x \) is less than \( y \)} \\
& x > y & \text{\( x \) is greater than \( y \)} \\
& x \geq y & \text{\( x \) is greater than or equal to \( y \)} \\
& x \leq y & \text{\( x \) is less than or equal to \( y \)}
\end{align*}
\]

Examples:

- EMPNO='528671'
- SALARY < 20000
- PRSTAFF <> :VAR1
- SALARY > (SELECT AVG(SALARY) FROM EMPLOYEE)
Notes:

1 The following forms of the comparison operators are also supported in basic and quantified predicates: ^=, ^<, ^>, !=, !<, and !>. In code pages 437, 819, and 850, the forms ¬=, ¬<, and ¬> are supported. All of these product-specific forms of the comparison operators are intended only to support existing SQL statements that use these operators, and are not recommended for use when writing new SQL statements.

A quantified predicate compares a value or values with a collection of values.

The fullselect must identify a number of columns that is the same as the number of expressions specified to the left of the predicate operator (SQLSTATE 428C4). The fullselect may return any number of rows.

When ALL is specified:
• The result of the predicate is true if the fullselect returns no values or if the specified relationship is true for every value returned by the fullselect.
• The result is false if the specified relationship is false for at least one value returned by the fullselect.
• The result is unknown if the specified relationship is not false for any of the values returned by the fullselect and at least one comparison is unknown because of the null value.

When SOME or ANY is specified:
• The result of the predicate is true if the specified relationship is true for each value of at least one row returned by the fullselect.
• The result is false if the fullselect returns no rows or if the specified relationship is false for at least one value of every row returned by the fullselect.
• The result is unknown if the specified relationship is not true for any of the rows and at least one comparison is unknown because of a null value.

Examples: Use the following tables when referring to the following examples.
Quantified predicate

Example 1

\[
\text{SELECT COLA FROM TBLAB}
\text{WHERE COLA = ANY(SELECT COLX FROM TBLXY)}
\]

Results in 2,3. The subselect returns (2,3). COLA in rows 2 and 3 equals at least one of these values.

Example 2

\[
\text{SELECT COLA FROM TBLAB}
\text{WHERE COLA > ANY(SELECT COLX FROM TBLXY)}
\]

Results in 3,4. The subselect returns (2,3). COLA in rows 3 and 4 is greater than at least one of these values.

Example 3

\[
\text{SELECT COLA FROM TBLAB}
\text{WHERE COLA > ALL(SELECT COLX FROM TBLXY)}
\]

Results in 4. The subselect returns (2,3). COLA in row 4 is the only one that is greater than both these values.

Example 4

\[
\text{SELECT COLA FROM TBLAB}
\text{WHERE COLA > ALL(SELECT COLX FROM TBLXY}
\text{WHERE COLX<0))}
\]

Results in 1,2,3,4, null. The subselect returns no values. Thus, the predicate is true for all rows in TBLAB.

Example 5

\[
\text{SELECT * FROM TBLAB}
\text{WHERE (COLA, COLB+10) = SOME (SELECT COLX, COLY FROM TBLXY)}
\]

The subselect returns all entries from TBLXY. The predicate is true for the subselect, hence the result is as follows:

<table>
<thead>
<tr>
<th>COLA</th>
<th>COLB</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>13</td>
</tr>
</tbody>
</table>

Example 6

\[
\text{SELECT * FROM TBLAB}
\text{WHERE (COLA, COLB) = ANY (SELECT COLX, COLY-10 FROM TBLXY)}
\]

The subselect returns COLX and COLY-10 from TBLXY. The predicate is true for the subselect, hence the result is as follows:
## Quantified predicate

<table>
<thead>
<tr>
<th>COLA</th>
<th>COLB</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>13</td>
</tr>
</tbody>
</table>
The ARRAY_EXISTS predicate tests for the existence of an array index in an array.

array-variable

An SQL variable, SQL parameter, or global variable of an array type, or a CAST specification of a parameter marker to an array type.

array-index

The data type of array-index must be assignable to the data type of the array index of the array. If array-variable is an ordinary array, then array-index must be assignable to INTEGER (SQLSTATE 428H1).

The result is true if array-variable includes an array index that is equal to array-index cast to the data type of the array index of array-variable; otherwise the result is false.

The result cannot be unknown; if either argument is null, the result is false.

Example

- Assume that array variable RECENT_CALLS is defined as an ordinary array of array type PHONENUMBERS. The following IF statement tests if the recent calls list has reached the 40th saved call yet. If it has, the local Boolean variable EIGHTY_PERCENT is set to true:

```
IF (ARRAY_EXISTS(RECENT_CALLS, 40)) THEN
  SET EIGHTY_PERCENT = TRUE;
END IF
```
The BETWEEN predicate compares a value with a range of values. If the data
types of the operands are not the same, all values are converted to the data type
that would result by applying the “Rules for result data types”, except if the data
types of all the operands are numeric, in which case no values are converted.

The BETWEEN predicate:

\[
\text{value1 \ BETWEEN \ value2 \ AND \ value3}
\]

is equivalent to the search condition:

\[
\text{value1} \geq \text{value2} \ \text{AND} \ \text{value1} \leq \text{value3}
\]

The BETWEEN predicate:

\[
\text{value1 \ NOT \ BETWEEN \ value2 \ AND \ value3}
\]

is equivalent to the search condition:

\[
\neg (\text{value1 \ BETWEEN \ value2 \ AND \ value3}); \text{that is,} \\
\text{value1} < \text{value2} \ \text{OR} \ \text{value1} > \text{value3}.
\]

The first operand (expression) cannot include a function that is not deterministic or
has an external action (SQLSTATE 42845).

**Examples**

*Example 1*

\[
\text{EMPLOYEE.SALARY \ BETWEEN \ 20000 \ AND \ 40000}
\]

Results in all salaries between $20,000.00 and $40,000.00.

*Example 2*

\[
\text{SALARY \ NOT \ BETWEEN \ 20000 + :HV1 \ AND \ 40000}
\]

Assuming :HV1 is 5000, results in all salaries below $25,000.00 and above
$40,000.00.
Cursor predicates

Cursor predicates are SQL keywords that can be used to determine the state of a cursor defined within the current scope. They provide a means for easily referencing whether a cursor is open, closed or if there are rows associated with the cursor.

`cursor-variable-name`

The name of a SQL variable or SQL parameter of a cursor type.

**IS**

Specifies that a cursor predicate property is to be tested.

**NOT**

Specifies that the opposite value of testing the cursor predicate property is to be returned.

**FOUND**

Specifies to check if the cursor contains rows after the execution of a FETCH statement. If the last FETCH statement executed was successful, and if the IS FOUND predicate syntax is used, the returned value is TRUE. If the last FETCH statement executed resulted in a condition where rows were not found, the result is false. The result is unknown when:

- the value of `cursor-variable-name` is null
- the underlying cursor of `cursor-variable-name` is not open
- the predicate is evaluated before the first FETCH action was performed on the underlying cursor
- the last FETCH action returned an error

The IS FOUND predicate can be useful within a portion of SQL PL logic that loops and performs a fetch with each iteration. The predicate can be used to determine if rows remain to be fetched. It provides an efficient alternative to using a condition handler that checks for the error condition that is raised when no more rows remain to be fetched.

When the NOT keyword is specified, so that the syntax is IS NOT FOUND, the result value is the opposite.

**OPEN**

Specifies to check if the cursor is in an open state. If the cursor is in open and if the IS OPEN predicate syntax is used, the returned value is TRUE. This can be a useful predicate in cases where cursors are passed as parameters to functions and procedures. Before attempting to open the cursor, this predicate can be used to determine if the cursor is already open.

When the NOT keyword is specified, so that the syntax is IS NOT OPEN, the result value is the opposite.

**Notes**

- A cursor predicate can only be used in statements within a compound SQL (compiled) statement (SQLSTATE 42818).
Example

The following script defines an SQL procedure that contains references to these predicates as well as the prerequisite objects required to successfully compile and call the procedure:

```sql
CREATE TABLE T1 (c1 INT, c2 INT, c3 INT)
INSERT INTO T1 VALUES (1,1,1),(2,2,2),(3,3,3)
CREATE TYPE myRowType AS ROW(c1 INT, c2 INT, c3 INT)
CREATE TYPE myCursorType AS myRowType CURSOR

CREATE PROCEDURE p(OUT count INT)
LANGUAGE SQL
BEGIN
    DECLARE c1 CURSOR;
    DECLARE lvarInt INT;
    SET count = -1;
    SET c1 = CURSOR FOR SELECT c1 FROM t1;
    IF (c1 IS NOT OPEN) THEN
        OPEN c1;
    ELSE
        set count = -2;
    END IF;
    SET count = 0;
    IF (c1 IS OPEN) THEN
        FETCH c1 INTO lvarInt;
        WHILE (c1 IS FOUND) DO
            SET count = count + 1;
            FETCH c1 INTO lvarInt;
        END WHILE;
    ELSE
        SET COUNT = 0;
    END IF;
END

CALL p()
```

Cursor predicates
**EXISTS predicate**

The EXISTS predicate tests for the existence of certain rows.

The fullselect may specify any number of columns, and
- The result is true only if the number of rows specified by the fullselect is not zero.
- The result is false only if the number of rows specified is zero
- The result cannot be unknown.

**Example**

```
EXISTS (SELECT * FROM TEMPL WHERE SALARY < 10000)
```
IN predicate

The IN predicate compares a value or values with a collection of values.

The fullselect must identify a number of columns that is the same as the number of expressions specified to the left of the IN keyword (SQLSTATE 428C4). The fullselect may return any number of rows.

- An IN predicate of the form:
  \[ expression \text{ IN } expression \]

  is equivalent to a basic predicate of the form:
  \[ expression = expression \]

- An IN predicate of the form:
  \[ expression \text{ IN } (\text{fullselect}) \]

  is equivalent to a quantified predicate of the form:
  \[ expression = \text{ANY } (\text{fullselect}) \]

- An IN predicate of the form:
  \[ expression \text{ NOT IN } (\text{fullselect}) \]

  is equivalent to a quantified predicate of the form:
  \[ expression <> \text{ALL } (\text{fullselect}) \]

- An IN predicate of the form:
  \[ (\text{expressiona, expressionb,..., expressionk}) \text{ IN } (\text{fullselect}) \]

  is equivalent to:
  \[ (\text{expressiona, expressionb,..., expressionk}) = \text{ANY } (\text{fullselect}) \]

  where fullselect in the values-clause form is:
  \[ \text{VALUES } (\text{expressiona}), (\text{expressionb}), ..., (\text{expressionk}) \]

- An IN predicate of the form:
  \[ (\text{expressiona, expressionb,..., expressionk}) \text{ IN } (\text{fullselect}) \]

  is equivalent to a quantified predicate of the form:
  \[ (\text{expressiona, expressionb,..., expressionk}) = \text{ANY } (\text{fullselect}) \]

Note that the operand on the left hand side of this form of these predicates is referred to as a row-value-expression.

The values for expression1 and expression2 or the column of fullselect1 in the IN predicate must be compatible. Each expression3 value and its corresponding column
of `fullselect2` in the IN predicate must be compatible. The rules for result data types can be used to determine the attributes of the result used in the comparison.

The values for the expressions in the IN predicate (including corresponding columns of a fullselect) can have different code pages. If a conversion is necessary, the code page is determined by applying rules for string conversions to the IN list first, and then to the predicate, using the derived code page for the IN list as the second operand.

**Examples**

*Example 1:* The following evaluates to true if the value in the row under evaluation in the DEPTNO column contains D01, B01, or C01:

```
DEPTNO IN ('D01', 'B01', 'C01')
```

*Example 2:* The following evaluates to true only if the EMPNO (employee number) on the left side matches the EMPNO of an employee in department E11:

```
EMPNO IN (SELECT EMPNO FROM EMPLOYEE WHERE WORKDEPT = 'E11')
```

*Example 3:* Given the following information, this example evaluates to true if the specific value in the row of the COL_1 column matches any of the values in the list:

<table>
<thead>
<tr>
<th>Expressions</th>
<th>Type</th>
<th>Code Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>COL_1</td>
<td>column</td>
<td>850</td>
</tr>
<tr>
<td>HV_2</td>
<td>host variable</td>
<td>437</td>
</tr>
<tr>
<td>HV_3</td>
<td>host variable</td>
<td>437</td>
</tr>
<tr>
<td>CON_1</td>
<td>constant</td>
<td>850</td>
</tr>
</tbody>
</table>

When evaluating the predicate:

```
COL_1 IN (:HV_2, :HV_3, CON_4)
```

the two host variables will be converted to code page 850, based on the rules for string conversions.

*Example 4:* The following evaluates to true if the specified year in EMENDATE (the date an employee activity on a project ended) matches any of the values specified in the list (the current year or the two previous years):

```
YEAR(EMENDATE) IN (YEAR(CURRENT DATE),
                   YEAR(CURRENT DATE - 1 YEAR),
                   YEAR(CURRENT DATE - 2 YEARS))
```

*Example 5:* The following evaluates to true if both ID and DEPT on the left side match MANAGER and DEPTNUMB respectively for any row of the ORG table.

```
(ID, DEPT) IN (SELECT MANAGER, DEPTNUMB FROM ORG)
```
LIKE predicate

The LIKE predicate searches for strings that have a certain pattern. The pattern is specified by a string in which the underscore and the percent sign may have special meanings. Trailing blanks in a pattern are part of the pattern.

If the value of any of the arguments is null, the result of the LIKE predicate is unknown.

The values for `match-expression`, `pattern-expression`, and `escape-expression` are compatible string expressions. There are slight differences in the types of string expressions supported for each of the arguments. The valid types of expressions are listed under the description of each argument.

None of the expressions can yield a distinct type. However, it can be a function that casts a distinct type to its source type.

**match-expression**
An expression that specifies the string that is to be examined to see if it conforms to a certain pattern of characters.

The expression can be specified by:
- A constant
- A special register
- A global variable
- A host variable (including a locator variable or a file reference variable)
- A scalar function
- A large object locator
- A column name
- An expression concatenating any of the above

**pattern-expression**
An expression that specifies the string that is to be matched.

The expression can be specified by:
- A constant
- A special register
- A global variable
- A host variable
- A scalar function whose operands are any of the above
- An expression concatenating any of the above
- An SQL procedure parameter

with the following restrictions:
- No element in the expression can be of type CLOB or DBCLOB. In addition it cannot be a BLOB file reference variable.
- The actual length of `pattern-expression` cannot be more than 32,672 bytes.
LIKE predicate

The following are examples of invalid string expressions or strings:
- SQL user-defined function parameters
- Trigger transition variables
- Local variables in dynamic compound statements

A simple description of the use of the LIKE predicate is that the pattern is used to specify the conformance criteria for values in the match-expression, where:
- The underscore character (_) represents any single character.
- The percent sign (%) represents a string of zero or more characters.
- Any other character represents itself.

If the pattern-expression needs to include either the underscore or the percent character, the escape-expression is used to specify a character to precede either the underscore or the percent character in the pattern.

A rigorous description of the use of the LIKE predicate follows. Note that this description ignores the use of the escape-expression; its use is covered later.
- Let \( m \) denote the value of match-expression and let \( p \) denote the value of pattern-expression. The string \( p \) is interpreted as a sequence of the minimum number of substring specifiers so each character of \( p \) is part of exactly one substring specifier. A substring specifier is an underscore, a percent sign, or any non-empty sequence of characters other than an underscore or a percent sign.

The result of the predicate is unknown if \( m \) or \( p \) is the null value. Otherwise, the result is either true or false. The result is true if \( m \) and \( p \) are both empty strings or there exists a partitioning of \( m \) into substrings such that:
  - A substring of \( m \) is a sequence of zero or more contiguous characters and each character of \( m \) is part of exactly one substring.
  - If the \( n \)th substring specifier is an underscore, the \( n \)th substring of \( m \) is any single character.
  - If the \( n \)th substring specifier is a percent sign, the \( n \)th substring of \( m \) is any sequence of zero or more characters.
  - If the \( n \)th substring specifier is neither an underscore nor a percent sign, the \( n \)th substring of \( m \) is equal to that substring specifier and has the same length as that substring specifier.
  - The number of substrings of \( m \) is the same as the number of substring specifiers.

Thus, if \( p \) is an empty string and \( m \) is not an empty string, the result is false. Similarly, it follows that if \( m \) is an empty string and \( p \) is not an empty string (except for a string containing only percent signs), the result is false.

The predicate \( m \) NOT LIKE \( p \) is equivalent to the search condition NOT (\( m \) LIKE \( p \)).

When the escape-expression is specified, the pattern-expression must not contain the escape character identified by the escape-expression, except when immediately followed by the escape character, the underscore character, or the percent sign character (SQLSTATE 22025).

If the match-expression is a character string in an MBCS database, it can contain mixed data. In this case, the pattern can include both SBCS and non-SBCS characters. For non-Unicode databases, the special characters in the pattern are interpreted as follows:
- An SBCS halfwidth underscore refers to one SBCS character.
A non-SBCS fullwidth underscore refers to one non-SBCS character.

An SBCS halfwidth or non-SBCS fullwidth percent sign refers to zero or more SBCS or non-SBCS characters.

In a Unicode database, there is really no distinction between "single-byte" and "non-single-byte" characters. Although the UTF-8 format is a "mixed-byte" encoding of Unicode characters, there is no real distinction between SBCS and non-SBCS characters in UTF-8. Every character is a Unicode character, regardless of the number of bytes in UTF-8 format.

In a Unicode graphic column, every non-supplementary character, including the halfwidth underscore character (U\'\005F') and the halfwidth percent sign character (U\'\0025'), is two bytes in width. In a Unicode database, special characters in a pattern are interpreted as follows:

- For character strings, a halfwidth underscore character (X'5F') or a fullwidth underscore character (X'EFBCBF') refers to one Unicode character, and a halfwidth percent sign character (X'25') or a fullwidth percent sign character (X'EFBC85') refers to zero or more Unicode characters.

- For graphic strings, a halfwidth underscore character (U\'\005F') or a fullwidth underscore character (U\'\FF3F') refers to one Unicode character, and a halfwidth percent sign character (U\'\0025') or a fullwidth percent sign character (U\'\FF05') refers to zero or more Unicode characters.

- To be recognized as special characters when a locale-sensitive UCA-based collation is in effect, the underscore character and the percent sign character must not be followed by non-spacing combining marks (diacritics). For example, the pattern U\'\%\0300' (percent sign character followed by non-spacing combining grave accent) will be interpreted as a search for % and not as a search for zero or more Unicode characters followed by a letter with a grave accent.

A Unicode supplementary character is stored as two graphic code points in a Unicode graphic column. To match a Unicode supplementary character in a Unicode graphic column, use one underscore if the database uses locale-sensitive UCA-based collation, and two underscores otherwise. To match a Unicode supplementary character in a Unicode character column, use one underscore for all collations. To match a base character with one or more trailing non-spacing combining characters, use one underscore if the database uses locale-sensitive UCA-based collation. Otherwise, use as many underscore characters as the number of non-spacing combining characters plus the base character.

escape-expression

This optional argument is an expression that specifies a character to be used to modify the special meaning of the underscore (_) and percent (%) characters in the pattern-expression. This allows the LIKE predicate to be used to match values that contain the actual percent and underscore characters.

The expression can be specified by any one of:

- A constant
- A special register
- A global variable
- A host variable
- A scalar function whose operands are any of the above
- An expression concatenating any of the above

with the restrictions that:
LIKE predicate

- No element in the expression can be of type CLOB or DBCLOB. In addition, it cannot be a BLOB file reference variable.
- For character columns, the result of the expression must be one character, or a binary string containing exactly one byte (SQLSTATE 22019).
- For graphic columns, the result of the expression must be one character (SQLSTATE 22019).
- The result of the expression must not be a non-spacing combining character sequence (for example, U&’\0301’, Combining Acute Accent).

When escape characters are present in the pattern string, an underscore, percent sign, or escape character can represent a literal occurrence of itself. This is true if the character in question is preceded by an odd number of successive escape characters. It is not true otherwise.

In a pattern, a sequence of successive escape characters is treated as follows:
- Let S be such a sequence, and suppose that S is not part of a larger sequence of successive escape characters. Suppose also that S contains a total of n characters. Then the rules governing S depend on the value of n:
  - If n is odd, S must be followed by an underscore or percent sign (SQLSTATE 22025). S and the character that follows it represent (n-1)/2 literal occurrences of the escape character followed by a literal occurrence of the underscore or percent sign.
  - If n is even, S represents n/2 literal occurrences of the escape character.
    Unlike the case where n is odd, S could end the pattern. If it does not end the pattern, it can be followed by any character (except, of course, an escape character, which would violate the assumption that S is not part of a larger sequence of successive escape characters). If S is followed by an underscore or percent sign, that character has its special meaning.

Following is an illustration of the effect of successive occurrences of the escape character which, in this case, is the back slash (\).

<table>
<thead>
<tr>
<th>Pattern string</th>
<th>Actual Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>%</td>
<td>A percent sign</td>
</tr>
<tr>
<td>\%</td>
<td>A back slash followed by zero or more arbitrary characters</td>
</tr>
<tr>
<td>\\%</td>
<td>A back slash followed by a percent sign</td>
</tr>
</tbody>
</table>

The code page used in the comparison is based on the code page of the
match-expression value.
- The match-expression value is never converted.
- If the code page of pattern-expression is different from the code page of
  match-expression, the value of pattern-expression is converted to the code page of
  match-expression, unless either operand is defined as FOR BIT DATA (in which case there is no conversion).
- If the code page of escape-expression is different from the code page of
  match-expression, the value of escape-expression is converted to the code page of
  match-expression, unless either operand is defined as FOR BIT DATA (in which case there is no conversion).
LIKE predicate

Notes

• The number of trailing blanks is significant in both the match-expression and the pattern-expression. If the strings are not the same length, the shorter string is not padded with blank spaces. For example, the expression 'PADDED ' LIKE 'PADDED' would not result in a match.

• If the pattern specified in a LIKE predicate is a parameter marker, and a fixed-length character host variable is used to replace the parameter marker, the value specified for the host variable must have the correct length. If the correct length is not specified, the select operation will not return the intended results.

For example, if the host variable is defined as CHAR(10), and the value WYSE% is assigned to that host variable, the host variable is padded with blanks on assignment. The pattern used is:

'WYSE%'

The database manager searches for all values that start with WYSE and that end with five blank spaces. If you want to search only for values that start with 'WYSE', assign a value of 'WSYE%%%%%' to the host variable.

• The pattern is matched using the collation of the database, unless either operand is defined as FOR BIT DATA, in which case the pattern is matched using a binary comparison.

Examples

• Search for the string 'SYSTEMS' appearing anywhere within the PROJNAME column in the PROJECT table.

`SELECT PROJNAME FROM PROJECT WHERE PROJECT.PROJNAME LIKE 'SYSTEMS%'`

• Search for a string with a first character of 'J' that is exactly two characters long in the FIRSTNME column of the EMPLOYEE table.

`SELECT FIRSTNME FROM EMPLOYEE WHERE EMPLOYEE.FIRSTNME LIKE 'J_'`

• Search for a string of any length, with a first character of 'J', in the FIRSTNME column of the EMPLOYEE table.

`SELECT FIRSTNME FROM EMPLOYEE WHERE EMPLOYEE.FIRSTNME LIKE 'J%'`

• In the CORP_SERVERS table, search for a string in the LA_SERVERS column that matches the value in the CURRENT SERVER special register.

`SELECT LA_SERVERS FROM CORP_SERVERS WHERE CORP_SERVERS.LA_SERVERS LIKE CURRENT SERVER`

• Retrieve all strings that begin with the character sequence '\' in column A of table T.

`SELECT A FROM T WHERE T.A LIKE '\_\_\_\_\_\_' ESCAPE '\'`

• Use the BLOB scalar function to obtain a one-byte escape character that is compatible with the match and pattern data types (both BLOBs).

`SELECT COLBLOB FROM TABLET WHERE COLBLOB LIKE :pattern_var ESCAPE BLOB(X'OE')`

• In a Unicode database defined with the case insensitive collation UCA500R1_LEN_S1, find all names that start with 'Bill'.

`SELECT NAME FROM CUSTDATA WHERE NAME LIKE 'Bill%'`

The will return the names 'Bill Smith', 'billy simon', and 'BILL JONES'.
The NULL predicate tests for null values.

The result of a NULL predicate cannot be unknown. If the value of the expression is null, the result is true. If the value is not null, the result is false. If NOT is specified, the result is reversed.

A row type value cannot be used as the operand in a NULL predicate, except as the qualifier of a field name. If expression is a row type, an error is returned (SQLSTATE 428H2).

Examples

**PHONENO IS NULL**

**SALARY IS NOT NULL**
A **TYPE predicate** compares the type of an expression with one or more user-defined structured types.

The dynamic type of an expression involving the dereferencing of a reference type is the actual type of the referenced row from the target typed table or view. This may differ from the target type of an expression involving the reference which is called the static type of the expression.

If the value of `expression` is null, the result of the predicate is unknown. The result of the predicate is true if the dynamic type of the `expression` is a subtype of one of the structured types specified by `typename`, otherwise the result is false. If `ONLY` precedes any `typename` the proper subtypes of that type are not considered.

If `typename` is not qualified, it is resolved using the SQL path. Each `typename` must identify a user-defined type that is in the type hierarchy of the static type of `expression` (SQLSTATE 428DU).

The `DEREF` function should be used whenever the TYPE predicate has an expression involving a reference type value. The static type for this form of `expression` is the target type of the reference.

The syntax IS OF and OF DYNAMIC TYPE are equivalent alternatives for the TYPE predicate. Similarly, IS NOT OF and NOT OF DYNAMIC TYPE are equivalent alternatives.

**Examples**

A table hierarchy exists with root table `EMPLOYEE` of type EMP and subtable `MANAGER` of type MGR. Another table, `ACTIVITIES`, includes a column called `WHO_RESPONSIBLE` that is defined as REF(EMP) SCOPE EMPLOYEE. The following is a type predicate that evaluates to true when a row corresponding to `WHO_RESPONSIBLE` is a manager:

```sql
DEREF (WHO_RESPONSIBLE) IS OF (MGR)
```

If a table contains a column `EMPLOYEE` of type EMP, `EMPLOYEE` may contain values of type EMP as well as values of its subtypes like MGR. The following predicate

```sql
EMPL IS OF (MGR)
```

returns true when EMPL is not null and is actually a manager.
The VALIDATED predicate tests whether or not the value specified by XML-expression has been validated using the XMLVALIDATE function. If the value specified is null, the result of the validation constraint is unknown; otherwise, the result of the validation constraint is either true or false. The value you specify must be of type XML.

If the ACCORDING TO XMLSCHEMA clause is not specified, then XML schemas used for validation do not impact the result of the validation constraint.

Description

XML-expression

Specifies the XML value tested, where XML-expression can consist of an XML document, XML content, a sequence of XML nodes, an XML column-name, or an XML correlation-name.

If an XML column-name is specified, the predicate evaluates whether or not XML documents associated with the specified column name have been validated.

See "CREATE TRIGGER" for information about specifying correlation names of type XML as part of triggers.

IS VALIDATED or IS NOT VALIDATED

Specifies the required validation state for the XML-expression operand.

For a constraint that specifies IS VALIDATED to evaluate as true, the operand must have been validated. If an optional ACCORDING TO XMLSCHEMA clause includes one or several XML schemas, the operand must have been validated using one of the identified XML schemas.

For a constraint that specifies IS NOT VALIDATED to evaluate as false, the operand must be in an validated state. If an optional ACCORDING TO XMLSCHEMA clause includes one or several XML schemas, the operand must have been validated using one of the identified XML schemas.

according-to-clause

Specifies one or several XML schemas against which the operand must or must
not have been validated. Only XML schemas previously registered with the XML schema repository may be specified.

**ACCORDING TO XMLSCHEMA**

**ID** XML-schema-name
Specifies an SQL identifier for the XML schema. The name, including the implicit or explicit SQL schema qualifier, must uniquely identify an existing XML schema in the XML schema repository at the current server. If no XML schema by this name exists in the implicitly or explicitly specified SQL schema, an error is returned (SQLSTATE 42704).

**URI** XML-uri1
Specifies the target namespace URI of the XML schema. The value of XML-uri1 specifies a URI as a character string constant that is not empty. The URI must be the target namespace of a registered XML schema (SQLSTATE 4274A) and, if no LOCATION clause is specified, it must uniquely identify the registered XML schema (SQLSTATE 4274B).

**NO NAMESPACE**
Specifies that the XML schema has no target namespace. The target namespace URI is equivalent to an empty character string that cannot be specified as an explicit target namespace URI.

**LOCATION** XML-uri2
Specifies the XML schema location URI of the XML schema. The value of XML-uri2 specifies a URI as a character string constant that is not empty. The XML schema location URI, combined with the target namespace URI, must identify a registered XML schema (SQLSTATE 4274A), and there must be only one such XML schema registered (SQLSTATE 4274B).

**Examples**

*Example 1:* Assume that column XMLCOL is defined in table T1. Retrieve only the XML values that have been validated by any XML schema.

```sql
SELECT XMLCOL FROM T1 WHERE XMLCOL IS VALIDATED
```

*Example 2:* Assume that column XMLCOL is defined in table T1. Enforce the rule that values cannot be inserted or updated unless they have been validated.

```sql
ALTER TABLE T1 ADD CONSTRAINT CK_VALIDATED
CHECK (XMLCOL IS VALIDATED)
```

*Example 3:* Assume that you want to select only those rows from table T1 with XML column XMLCOL that have been validated with the XML schema URI http://www.posample.org.

```sql
SELECT XMLCOL FROM T1 WHERE XMLCOL IS VALIDATED
   ACCORDING TO XMLSCHEMA URI
   'http://www.posample.org'
```
XMLEXISTS predicate

Notes:
1 The data type cannot be DECFLOAT.
2 The data type of the expression cannot be DECFLOAT.

The XMLEXISTS predicate tests whether an XQuery expression returns a sequence of one or more items.

**xquery-expression-constant**
Specifies an SQL character string constant that is interpreted as an XQuery expression. The constant string is converted directly to UTF-8 without conversion to the database or section code page. The XQuery expression executes using an optional set of input XML values, and returns an output sequence that is tested to determine the result of the XMLEXISTS predicate. The value for *xquery-expression-constant* must not be an empty string or a string of blank characters (SQLSTATE 10505).

**PASSING**
Specifies input values and the manner in which these values are passed to the XQuery expression specified by *xquery-expression-constant*. By default, every unique column name that is in the scope where the function is invoked is implicitly passed to the XQuery expression using the name of the column as the variable name. If an identifier in a specified xquery-argument matches an in-scope column name, then the explicit xquery-argument is passed to the XQuery expression overriding that implicit column.

**BY REF**
Specifies that the default passing mechanism is by reference for any *xquery-variable-expression* of data type XML. When XML values are passed by reference, the XQuery evaluation uses the input node trees, if any, directly from the specified input expressions, preserving all properties, including the original node identities and document order. If two arguments pass the same XML value, node identity comparisons and document ordering comparisons involving some nodes contained between the two input arguments might refer to nodes within the same XML node tree.

This clause has no impact on how non-XML values are passed. The non-XML values create a new copy of the value during the cast to XML.

**xquery-argument**
Specifies an argument that is to be passed to the XQuery expression
XMLEXISTS predicate

specified by xquery-expression-constant. An argument specifies a value and the manner in which that value is to be passed. The argument includes an SQL expression that is evaluated.

- If the resulting value is of type XML, it becomes an input-xml-value. A null XML value is converted to an XML empty sequence.
- If the resulting value is not of type XML, it must be castable to the XML data type. A null value is converted to an XML empty sequence. The converted value becomes an input-xml-value.

When the xquery-expression-constant is evaluated, an XQuery variable is presented with a value equal to input-xml-value and a name specified by the AS clause.

xquery-variable-expression

Specifies an SQL expression whose value is available to the XQuery expression specified by xquery-expression-constant during execution. The expression cannot contain a sequence reference (SQLSTATE 428F9) or an OLAP function (SQLSTATE 42903). The data type of the expression cannot be DECFLOAT.

AS identifier

Specifies that the value generated by xquery-variable-expression will be passed to xquery-expression-constant as an XQuery variable. The variable name will be identifier. The leading dollar sign ($) that precedes variable names in the XQuery language is not included in identifier. The identifier must be a valid XQuery variable name and is restricted to an XML NCName. The identifier must not be greater than 128 bytes in length. Two arguments within the same PASSING clause cannot use the same identifier (SQLSTATE 42711).

BY REF

Indicates that an XML input value is to be passed by reference. When XML values are passed by reference, the XQuery evaluation uses the input node trees, if any, directly from the specified input expressions, preserving all properties, including the original node identities and document order. If two arguments pass the same XML value, node identity comparisons and document ordering comparisons involving some nodes contained between the two input arguments might refer to nodes within the same XML node tree. If BY REF is not specified following an xquery-variable-expression, XML arguments are passed by way of the default passing mechanism that is provided through the syntax that follows the PASSING keyword. This option cannot be specified for non-XML values. When a non-XML value is passed, the value is converted to XML; this process creates a copy.

Notes

The XMLEXISTS predicate cannot be:

- Part of the ON clause that is associated with a JOIN operator or a MERGE statement (SQLSTATE 42972)
- Part of the GENERATE KEY USING or RANGE THROUGH clause in the CREATE INDEX EXTENSION statement (SQLSTATE 428E3)
- Part of the FILTER USING clause in the CREATE FUNCTION (External Scalar) statement, or the FILTER USING clause in the CREATE INDEX EXTENSION statement (SQLSTATE 428E4)
- Part of a check constraint or a column generation expression (SQLSTATE 42621)
**XMLEXISTS predicate**

- Part of a group-by-clause (SQLSTATE 42822)
- Part of an argument for a column-function (SQLSTATE 42607)

An XMLEXISTS predicate that involves a subquery might be restricted by statements that restrict subqueries.

**Example**

```sql
SELECT c.cid FROM customer c
WHERE XMLEXISTS('{$d/*:customerinfo/*:addr[ *:city = "Aurora"]' PASSING info AS "d")
```
Chapter 3. Functions

Functions overview

A function is an operation that is denoted by a function name followed by a pair of parentheses enclosing the specification of arguments (there may be no arguments).

Built-in functions are provided with the database manager; they return a single result value, and are identified as part of the SYSIBM schema. Built-in functions include column functions (such as AVG), operator functions (such as “+”), casting functions (such as DECIMAL), and others (such as SUBSTR).

User-defined functions are registered to a database in SYSCAT.ROUTINES (using the CREATE FUNCTION statement). User-defined functions are never part of the SYSIBM schema. One such set of functions is provided with the database manager in a schema called SYSFUN, and another in a schema called SYSPROC.

Functions are classified as aggregate (column) functions, scalar functions, row functions, or table functions.

- The argument of an aggregate function is a collection of like values. An aggregate function returns a single value (possibly null), and can be specified in an SQL statement wherever an expression can be used.
- The arguments of a scalar function are individual scalar values, which can be of different types and have different meanings. A scalar function returns a single value (possibly null), and can be specified in an SQL statement wherever an expression can be used.
- The argument of a row function is a structured type. A row function returns a row of built-in data types and can only be specified as a transform function for a structured type.
- The arguments of a table function are individual scalar values, which can be of different types and have different meanings. A table function returns a table to the SQL statement, and can be specified only within the FROM clause of a SELECT statement.

The function name, combined with the schema, gives the fully qualified name of a function. The combination of schema, function name, and input parameters make up a function signature.

In some cases, the input parameter type is specified as a specific built-in data type, and in other cases, it is specified through a general variable like any-numeric-type. If a particular data type is specified, an exact match will only occur with the specified data type. If a general variable is used, each of the data types associated with that variable results in an exact match.

Additional functions may be available, because user-defined functions can be created in different schemas, using one of the function signatures as a source. You can also create external functions in your applications.
Supported functions and administrative SQL routines and views

This topic lists the supported built-in functions classified by type:

- Aggregate functions (Table 32)
- Array functions (Table 33 on page 305)
- Cast scalar functions (Table 34 on page 306)
- Datetime scalar functions (Table 35 on page 307)
- Miscellaneous scalar functions (Table 36 on page 308)
- Numeric scalar functions (Table 37 on page 309)
- Partitioning scalar functions (Table 38 on page 311)
- Security scalar functions (Table 39 on page 311)
- String scalar functions (Table 40 on page 311)
- Table functions (Table 41 on page 313)
- XML functions (Table 42 on page 313)

For lists of the supported administrative SQL routines and views classified by functionality, see “Supported administrative SQL routines and views” in Administrative Routines and Views. These routines and views are grouped as follows:

- Activity monitor administrative SQL routines
- ADMIN_CMD stored procedure and associated administrative SQL routines
- Configuration administrative SQL routines and views
- Environment administrative views
- Health snapshot administrative SQL routines
- MQSeries® administrative SQL routines
- Security administrative SQL routines and views
- Snapshot administrative SQL routines and views
- SQL procedures administrative SQL routines
- Stepwise redistribute administrative SQL routines
- Storage management tool administrative SQL routines
- Miscellaneous administrative SQL routines and views

Table 32. Aggregate functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>“ARRAY_AGG” on page 316</td>
<td>Aggregates a set of elements into an array.</td>
</tr>
<tr>
<td>“AVG” on page 318</td>
<td>Returns the average of a set of numbers.</td>
</tr>
<tr>
<td>“CORRELATION” on page 320</td>
<td>Returns the coefficient of correlation of a set of number pairs.</td>
</tr>
<tr>
<td>“COUNT” on page 321</td>
<td>Returns the number of rows or values in a set of rows or values.</td>
</tr>
<tr>
<td>“COUNT_BIG” on page 322</td>
<td>Returns the number of rows or values in a set of rows or values. The result can be greater than the maximum value of INTEGER.</td>
</tr>
<tr>
<td>“COVARIANCE” on page 324</td>
<td>Returns the covariance of a set of number pairs.</td>
</tr>
</tbody>
</table>
## Supported functions and administrative SQL routines and views

### Table 32. Aggregate functions (continued)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>“GROUPING” on page 325</td>
<td>Used with grouping-sets and super-groups to indicate sub-total rows generated by a grouping set. The value returned is 0 or 1. A value of 1 means that the value of the argument in the returned row is a null value, and the row was generated for a grouping set. This generated row provides a sub-total for a grouping set.</td>
</tr>
<tr>
<td>“MAX” on page 327</td>
<td>Returns the maximum value in a set of values.</td>
</tr>
<tr>
<td>“MIN” on page 328</td>
<td>Returns the minimum value in a set of values.</td>
</tr>
<tr>
<td>“Regression functions” on page 329</td>
<td>The REGR_AVGX aggregate function returns quantities used to compute diagnostic statistics.</td>
</tr>
<tr>
<td>“Regression functions” on page 329</td>
<td>The REGR_AVGY aggregate function returns quantities used to compute diagnostic statistics.</td>
</tr>
<tr>
<td>“Regression functions” on page 329</td>
<td>The REGR_COUNT aggregate function returns the number of non-null number pairs used to fit the regression line.</td>
</tr>
<tr>
<td>“Regression functions” on page 329</td>
<td>The REGR_INTERCEPT or REGR_ICPT aggregate function returns the y-intercept of the regression line.</td>
</tr>
<tr>
<td>“Regression functions” on page 329</td>
<td>The REGR_R2 aggregate function returns the coefficient of determination for the regression.</td>
</tr>
<tr>
<td>“Regression functions” on page 329</td>
<td>The REGR_SLOPE aggregate function returns the slope of the line.</td>
</tr>
<tr>
<td>“Regression functions” on page 329</td>
<td>The REGR_SXX aggregate function returns quantities used to compute diagnostic statistics.</td>
</tr>
<tr>
<td>“Regression functions” on page 329</td>
<td>The REGR_SXY aggregate function returns quantities used to compute diagnostic statistics.</td>
</tr>
<tr>
<td>“Regression functions” on page 329</td>
<td>The REGR_SYX aggregate function returns quantities used to compute diagnostic statistics.</td>
</tr>
<tr>
<td>“STDDEV” on page 332</td>
<td>Returns the standard deviation of a set of numbers.</td>
</tr>
<tr>
<td>“SUM” on page 333</td>
<td>Returns the sum of a set of numbers.</td>
</tr>
<tr>
<td>“VARIANCE” on page 334</td>
<td>Returns the variance of a set of numbers.</td>
</tr>
<tr>
<td>“XMLAGG” on page 335</td>
<td>Returns an XML sequence containing an item for each non-null value in a set of XML values.</td>
</tr>
<tr>
<td>“XMLGROUP” on page 337</td>
<td>Returns an XML value with a single XQuery document node containing one top-level element node.</td>
</tr>
</tbody>
</table>

### Table 33. Array functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>“ARRAY_AGG” on page 316</td>
<td>Aggregates a set of elements into an array.</td>
</tr>
<tr>
<td>“ARRAY_DELETE” on page 343</td>
<td>Deletes an element or range of elements from an associative array.</td>
</tr>
<tr>
<td>“ARRAY_FIRST” on page 346</td>
<td>Returns the smallest array index value of the array.</td>
</tr>
<tr>
<td>“ARRAY_LAST” on page 347</td>
<td>Returns the largest array index value of the array.</td>
</tr>
<tr>
<td>“ARRAY_NEXT” on page 348</td>
<td>Returns the next larger array index value for an array relative to the specified array index argument.</td>
</tr>
<tr>
<td>“ARRAY_PRIOR” on page 349</td>
<td>Returns the next smaller array index value for an array relative to the specified array index argument.</td>
</tr>
</tbody>
</table>
### Supported functions and administrative SQL routines and views

#### Table 33. Array functions (continued)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>“CARDINALITY” on page 360</td>
<td>Returns a value of type BIGINT representing the number of elements of an array.</td>
</tr>
<tr>
<td>“MAX_CARDINALITY” on page 477</td>
<td>Returns a value of type BIGINT representing the maximum number of elements that an array can contain.</td>
</tr>
<tr>
<td>“TRIM_ARRAY” on page 590</td>
<td>Returns a value with the same array type as array-variable but with the cardinality reduced by the value of numeric-expression.</td>
</tr>
<tr>
<td>“UNNEST” on page 671</td>
<td>Returns a result table that includes a row for each element of the specified array.</td>
</tr>
</tbody>
</table>

#### Table 34. Cast scalar functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>“BIGINT” on page 355</td>
<td>Returns a 64-bit integer representation of a value in the form of an integer constant.</td>
</tr>
<tr>
<td>“BLOB” on page 359</td>
<td>Returns a BLOB representation of a string of any type.</td>
</tr>
<tr>
<td>“CHAR” on page 362</td>
<td>Returns a CHARACTER representation of a value.</td>
</tr>
<tr>
<td>“CLOB” on page 371</td>
<td>Returns a CLOB representation of a value.</td>
</tr>
<tr>
<td>“DATE” on page 383</td>
<td>Returns a DATE from a value.</td>
</tr>
<tr>
<td>“DBCLOB” on page 390</td>
<td>Returns a DBCLOB representation of a string.</td>
</tr>
<tr>
<td>“DECIMAL or DEC” on page 397</td>
<td>Returns the decimal floating-point representation of a value.</td>
</tr>
<tr>
<td>“EMPTY_BLOB”</td>
<td>Returns the floating-point representation of a value.</td>
</tr>
<tr>
<td>“EMPTY_CLOB”</td>
<td>Returns a zero-length value of the associated data type.</td>
</tr>
<tr>
<td>“EMPTY_DBCLOB”</td>
<td>Returns a DECIMAL representation of a value.</td>
</tr>
<tr>
<td>“FLOAT” on page 419</td>
<td>Returns a FLOAT representation of a value.</td>
</tr>
<tr>
<td>“GRAPHIC” on page 424</td>
<td>Returns a GRAPHIC representation of a string.</td>
</tr>
<tr>
<td>“INTEGER or INT” on page 446</td>
<td>Returns an INTEGER representation of a value.</td>
</tr>
<tr>
<td>“REAL” on page 512</td>
<td>Returns the single-precision floating-point representation of a value.</td>
</tr>
<tr>
<td>“SMALLINT” on page 547</td>
<td>Returns a SMALLINT representation of a value.</td>
</tr>
<tr>
<td>“TIME” on page 565</td>
<td>Returns a TIME from a value.</td>
</tr>
<tr>
<td>“TIMESTAMP” on page 566</td>
<td>Returns a TIMESTAMP from a value or a pair of values.</td>
</tr>
<tr>
<td>“TO_CLOB” on page 579</td>
<td>Returns a CLOB representation of a character string type.</td>
</tr>
<tr>
<td>“VARCHAR” on page 605</td>
<td>Returns a VARCHAR representation of a value.</td>
</tr>
<tr>
<td>“VARGRAPHIC” on page 620</td>
<td>Returns a VARGRAPHIC representation of a value.</td>
</tr>
</tbody>
</table>
## Supported functions and administrative SQL routines and views

### Table 35. Datetime scalar functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;ADD_MONTHS&quot; on page 343</td>
<td>Returns a datetime value that represents expression plus a specified number of months.</td>
</tr>
<tr>
<td>&quot;DAY&quot; on page 384</td>
<td>Returns the day part of a value.</td>
</tr>
<tr>
<td>&quot;DAYNAME&quot; on page 385</td>
<td>Returns a character string containing the name of the day (for example, Friday) for the day portion of expression, based on locale-name or the value of the special register CURRENT LOCALE LC_TIME.</td>
</tr>
<tr>
<td>&quot;DAYOFWEEK&quot; on page 386</td>
<td>Returns the day of the week from a value, where 1 is Sunday and 7 is Saturday.</td>
</tr>
<tr>
<td>&quot;DAYOFWEEK_ISO&quot; on page 387</td>
<td>Returns the day of the week from a value, where 1 is Monday and 7 is Sunday.</td>
</tr>
<tr>
<td>&quot;DAYOFYEAR&quot; on page 388</td>
<td>Returns the day of the year from a value.</td>
</tr>
<tr>
<td>&quot;DAYS&quot; on page 389</td>
<td>Returns an integer representation of a date.</td>
</tr>
<tr>
<td>&quot;EXTRACT&quot; on page 417</td>
<td>Returns a portion of a date or timestamp based on the arguments.</td>
</tr>
<tr>
<td>&quot;HOUR&quot; on page 434</td>
<td>Returns the hour part of a value.</td>
</tr>
<tr>
<td>&quot;JULIAN_DAY&quot; on page 448</td>
<td>Returns an integer value representing the number of days from January 1, 4712 B.C. to the date specified in the argument.</td>
</tr>
<tr>
<td>&quot;LAST_DAY&quot; on page 449</td>
<td>Returns a datetime value that represents the last day of the month of the argument.</td>
</tr>
<tr>
<td>&quot;MICROSECOND&quot; on page 478</td>
<td>Returns the microsecond part of a value.</td>
</tr>
<tr>
<td>&quot;MIDNIGHT_SECONDS&quot; on page 479</td>
<td>Returns an integer value representing the number of seconds between midnight and a specified time value.</td>
</tr>
<tr>
<td>&quot;MINUTE&quot; on page 481</td>
<td>Returns the minute part of a value.</td>
</tr>
<tr>
<td>&quot;MONTH&quot; on page 483</td>
<td>Returns the month part of a value.</td>
</tr>
<tr>
<td>&quot;MONTHNAME&quot; on page 484</td>
<td>Returns a character string containing the name of the month (for example, January) for the month portion of expression, based on locale-name or the value of the special register CURRENT LOCALE LC_TIME.</td>
</tr>
<tr>
<td>&quot;MONTHS_BETWEEN&quot; on page 485</td>
<td>Returns an estimate of the number of months between expression1 and expression2.</td>
</tr>
<tr>
<td>&quot;NEXT_DAY&quot; on page 489</td>
<td>Returns a datetime value that represents the first weekday, named by string-expression, that is later than the date in expression.</td>
</tr>
<tr>
<td>&quot;QUARTER&quot; on page 508</td>
<td>Returns an integer that represents the quarter of the year in which a date resides.</td>
</tr>
<tr>
<td>&quot;ROUND&quot; on page 526</td>
<td>Returns a datetime value, rounded to the unit specified by format-string.</td>
</tr>
<tr>
<td>&quot;ROUND_TIMESTAMP&quot; on page 532</td>
<td>Returns a timestamp that is the expression rounded to the unit specified by the format-string.</td>
</tr>
<tr>
<td>&quot;SECOND&quot; on page 542</td>
<td>Returns the seconds part of a value.</td>
</tr>
<tr>
<td>&quot;TIMESTAMP_FORMAT&quot; on page 568</td>
<td>Returns a timestamp from a character string (argument1) that has been interpreted using a format template (argument2).</td>
</tr>
</tbody>
</table>
### Table 35. Datetime scalar functions (continued)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;TIMESTAMP_ISO&quot; on page 575</td>
<td>Returns a timestamp value based on a date, time, or timestamp argument. If the argument is a date, it inserts zero for all the time elements. If the argument is a time, it inserts the value of CURRENT DATE for the date elements, and zero for the fractional time element.</td>
</tr>
<tr>
<td>&quot;TIMESTAMPDIFF&quot; on page 576</td>
<td>Returns an estimated number of intervals of type argument1, based on the difference between two timestamps. The second argument is the result of subtracting two timestamp types and converting the result to CHAR.</td>
</tr>
<tr>
<td>&quot;TO_CHAR&quot; on page 578</td>
<td>Returns a CHARACTER representation of a timestamp.</td>
</tr>
<tr>
<td>&quot;TO_DATE&quot; on page 580</td>
<td>Returns a timestamp from a character string.</td>
</tr>
<tr>
<td>&quot;TO_TIMESTAMP&quot; on page 582</td>
<td>Returns a timestamp that is based on the interpretation of the input string using the specified format.</td>
</tr>
<tr>
<td>&quot;TRUNCATE or TRUNC&quot; on page 593</td>
<td>Returns a datetime value, truncated to the unit specified by format-string.</td>
</tr>
<tr>
<td>&quot;TRUNC_TIMESTAMP&quot; on page 591</td>
<td>Returns a timestamp that is the expression truncated to the unit specified by the format-string.</td>
</tr>
<tr>
<td>&quot;VARCHAR_FORMAT&quot; on page 611</td>
<td>Returns a CHARACTER representation of a timestamp (argument1), formatted according to a template (argument2).</td>
</tr>
<tr>
<td>&quot;WEEK&quot; on page 626</td>
<td>Returns the week of the year from a value, where the week starts with Sunday.</td>
</tr>
<tr>
<td>&quot;WEEK_ISO&quot; on page 627</td>
<td>Returns the week of the year from a value, where the week starts with Monday.</td>
</tr>
<tr>
<td>&quot;YEAR&quot; on page 668</td>
<td>Returns the year part of a value.</td>
</tr>
</tbody>
</table>

### Table 36. Miscellaneous scalar functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;BITAND, BITANDNOT, BITOR, BITXOR, and BITNOT&quot; on page 357</td>
<td>These bitwise functions operate on the &quot;two's complement&quot; representation of the integer value of the input arguments and return the result as a corresponding base 10 integer value in a data type based on the data type of the input arguments.</td>
</tr>
<tr>
<td>&quot;COALESCE&quot; on page 372</td>
<td>Returns the first argument that is not null.</td>
</tr>
<tr>
<td>&quot;CURSOR_ROWCOUNT&quot; on page 381</td>
<td>Returns the cumulative count of all rows fetched by the specified cursor since the cursor was opened.</td>
</tr>
<tr>
<td>&quot;DECODE&quot; on page 401</td>
<td>Compares each specified expression2 to expression1. If expression1 is equal to expression2, or both expression1 and expression2 are null, the value of the following result-expression is returned. If no expression2 matches expression1, the value of else-expression is returned; otherwise a null value is returned.</td>
</tr>
<tr>
<td>&quot;DEREF&quot; on page 406</td>
<td>Returns an instance of the target type of the reference type argument.</td>
</tr>
<tr>
<td>&quot;EVENT_MON_STATE&quot; on page 415</td>
<td>Returns the operational state of particular event monitor.</td>
</tr>
<tr>
<td>&quot;GREATEST&quot; on page 429</td>
<td>Returns the maximum value in a set of values.</td>
</tr>
<tr>
<td>&quot;HEX&quot; on page 432</td>
<td>Returns a hexadecimal representation of a value.</td>
</tr>
</tbody>
</table>
Table 36. Miscellaneous scalar functions (continued)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;IDENTITY_VAL_LOCAL&quot;</td>
<td>on page 435</td>
</tr>
<tr>
<td>&quot;LEAST&quot; on page 452</td>
<td>Returns the minimum value in a set of values.</td>
</tr>
<tr>
<td>&quot;LENGTH&quot; on page 456</td>
<td>Returns the length of a value.</td>
</tr>
<tr>
<td>&quot;MAX&quot; on page 476</td>
<td>Returns the maximum value in a set of values.</td>
</tr>
<tr>
<td>&quot;MIN&quot; on page 480</td>
<td>Returns the minimum value in a set of values.</td>
</tr>
<tr>
<td>&quot;NULLIF&quot; on page 492</td>
<td>Returns a null value if the arguments are equal; otherwise, it returns the value of the first argument.</td>
</tr>
<tr>
<td>&quot;NVL&quot; on page 493</td>
<td>Returns the first argument that is not null.</td>
</tr>
<tr>
<td>&quot;RAISE_ERROR&quot; on page 510</td>
<td>Raises an error in the SQLCA. The sqlstate that is to be returned is indicated by argument1. The second argument contains any text that is to be returned.</td>
</tr>
<tr>
<td>&quot;REC2XML&quot; on page 514</td>
<td>Returns a string formatted with XML tags, containing column names and column data.</td>
</tr>
<tr>
<td>&quot;RID_BIT and RID&quot; on page 521</td>
<td>The RID_BIT scalar function returns the row identifier (RID) of a row in a character string format. The RID scalar function returns the RID of a row in large integer format. The RID function is not supported in partitioned database environments. The RID_BIT function is preferred over the RID function.</td>
</tr>
<tr>
<td>&quot;TABLE_NAME&quot; on page 560</td>
<td>Returns an unqualified name of a table or view based on the object name specified in argument1, and the optional schema name specified in argument2. The returned value is used to resolve aliases.</td>
</tr>
<tr>
<td>&quot;TABLE_SCHEMA&quot; on page 561</td>
<td>Returns the schema name portion of a two-part table or view name (given by the object name in argument1 and the optional schema name in argument2). The returned value is used to resolve aliases.</td>
</tr>
<tr>
<td>&quot;TYPE_ID&quot; on page 596</td>
<td>Returns the internal data type identifier of the dynamic data type of the argument. The result of this function is not portable across databases.</td>
</tr>
<tr>
<td>&quot;TYPE_NAME&quot; on page 597</td>
<td>Returns the unqualified name of the dynamic data type of the argument.</td>
</tr>
<tr>
<td>&quot;TYPE_SCHEMA&quot; on page 598</td>
<td>Returns the schema name of the dynamic data type of the argument.</td>
</tr>
<tr>
<td>&quot;VALUE&quot; on page 604</td>
<td>Returns the first argument that is not null.</td>
</tr>
</tbody>
</table>

Table 37. Numeric scalar functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;ABS or ABSVAL&quot; on page 341</td>
<td>Returns the absolute value of a number.</td>
</tr>
<tr>
<td>&quot;ACOS&quot; on page 342</td>
<td>Returns the arc cosine of a number, in radians.</td>
</tr>
<tr>
<td>&quot;ASIN&quot; on page 351</td>
<td>Returns the arc sine of a number, in radians.</td>
</tr>
<tr>
<td>&quot;ATAN&quot; on page 352</td>
<td>Returns the arc tangent of a number, in radians.</td>
</tr>
<tr>
<td>&quot;ATANH&quot; on page 354</td>
<td>Returns the hyperbolic arc tangent of a number, in radians.</td>
</tr>
<tr>
<td>&quot;ATAN2&quot; on page 353</td>
<td>Returns the arc tangent of x and y coordinates as an angle expressed in radians.</td>
</tr>
</tbody>
</table>
### Table 37. Numeric scalar functions (continued)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>“CEILING or CEIL” on page 361</td>
<td>Returns the smallest integer value that is greater than or equal to a number.</td>
</tr>
<tr>
<td>“COMPARE_DECFLOAT” on page 375</td>
<td>Returns a SMALLINT value that indicates whether the two arguments are equal or unordered, or whether one argument is greater than the other.</td>
</tr>
<tr>
<td>“COS” on page 378</td>
<td>Returns the cosine of a number.</td>
</tr>
<tr>
<td>“COSH” on page 379</td>
<td>Returns the hyperbolic cosine of a number.</td>
</tr>
<tr>
<td>“COT” on page 380</td>
<td>Returns the cotangent of the argument, where the argument is an angle expressed in radians.</td>
</tr>
<tr>
<td>“DECFLOAT_FORMAT” on page 395</td>
<td>Returns a DECFLOAT(34) from a character string.</td>
</tr>
<tr>
<td>“DEGREES” on page 405</td>
<td>Returns the number of degrees of an angle.</td>
</tr>
<tr>
<td>“DIGITS” on page 408</td>
<td>Returns a character-string representation of the absolute value of a number.</td>
</tr>
<tr>
<td>“EXP” on page 416</td>
<td>Returns a value that is the base of the natural logarithm (e) raised to a power specified by the argument.</td>
</tr>
<tr>
<td>“FLOOR” on page 420</td>
<td>Returns the largest integer value that is less than or equal to a number.</td>
</tr>
<tr>
<td>“LN” on page 458</td>
<td>Returns the natural logarithm of a number.</td>
</tr>
<tr>
<td>“LOG10” on page 466</td>
<td>Returns the common logarithm (base 10) of a number.</td>
</tr>
<tr>
<td>“MOD” on page 482</td>
<td>Returns the remainder of the first argument divided by the second argument.</td>
</tr>
<tr>
<td>“MULTIPLY_ALT” on page 487</td>
<td>Returns the product of two arguments as a decimal value. This function is useful when the sum of the argument precisions is greater than 31.</td>
</tr>
<tr>
<td>“NORMALIZE_DECFLOAT” on page 491</td>
<td>Returns a decimal floating-point value that is the result of the argument set to its simplest form.</td>
</tr>
<tr>
<td>“POWER” on page 505</td>
<td>Returns the result of raising the first argument to the power of the second argument.</td>
</tr>
<tr>
<td>“QUANTIZE” on page 506</td>
<td>Returns a decimal floating-point number that is equal in value and sign to the first argument, and whose exponent is equal to the exponent of the second argument.</td>
</tr>
<tr>
<td>“RADIANS” on page 509</td>
<td>Returns the number of radians for an argument that is expressed in degrees.</td>
</tr>
<tr>
<td>“RAND” on page 511</td>
<td>Returns a random number.</td>
</tr>
<tr>
<td>“ROUND” on page 526</td>
<td>Returns a numeric value that has been rounded to the specified number of decimal places.</td>
</tr>
<tr>
<td>“SIGN” on page 544</td>
<td>Returns the sign of a number.</td>
</tr>
<tr>
<td>“SIN” on page 545</td>
<td>Returns the sine of a number.</td>
</tr>
<tr>
<td>“SINH” on page 546</td>
<td>Returns the hyperbolic sine of a number.</td>
</tr>
<tr>
<td>“SQRT” on page 550</td>
<td>Returns the square root of a number.</td>
</tr>
<tr>
<td>“TAN” on page 563</td>
<td>Returns the tangent of a number.</td>
</tr>
<tr>
<td>“TANH” on page 564</td>
<td>Returns the hyperbolic tangent of a number.</td>
</tr>
<tr>
<td>“TO_NUMBER” on page 581</td>
<td>Returns a DECFLOAT(34) from a character string.</td>
</tr>
</tbody>
</table>
### Table 37. Numeric scalar functions (continued)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;TOTALORDER&quot; on page 583</td>
<td>Returns a SMALLINT value of -1, 0, or 1 that indicates the comparison order of two arguments.</td>
</tr>
<tr>
<td>&quot;TRUNCATE or TRUNC&quot; on page 593</td>
<td>Returns a number value that has been truncated at a specified number of decimal places.</td>
</tr>
</tbody>
</table>

### Table 38. Partitioning scalar functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;DATAPARTITIONNUM&quot; on page 382</td>
<td>Returns the sequence number (SYSDATAPARTITIONS.SEQNO) of the data partition in which the row resides. The argument is any column name within the table.</td>
</tr>
<tr>
<td>&quot;DBPARTITIONNUM&quot; on page 391</td>
<td>Returns the database partition number of the row. The argument is any column name within the table.</td>
</tr>
<tr>
<td>&quot;HASHEDVALUE&quot; on page 430</td>
<td>Returns the distribution map index (0 to 32767) of the row. The argument is a column name within a table.</td>
</tr>
</tbody>
</table>

### Table 39. Security scalar functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;SECLABEL&quot; on page 538</td>
<td>Returns an unnamed security label.</td>
</tr>
<tr>
<td>&quot;SECLABEL_BY_NAME&quot; on page 539</td>
<td>Returns a specific security label.</td>
</tr>
<tr>
<td>&quot;SECLABEL_TO_CHAR&quot; on page 540</td>
<td>Accepts a security label and returns a string that contains all elements in the security label.</td>
</tr>
</tbody>
</table>

### Table 40. String scalar functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;ASCII&quot; on page 350</td>
<td>Returns the ASCII code value of the leftmost character of the argument as an integer.</td>
</tr>
<tr>
<td>&quot;CHARACTER_LENGTH&quot; on page 368</td>
<td>Returns the length of an expression in the specified string-unit.</td>
</tr>
<tr>
<td>&quot;CHR&quot; on page 370</td>
<td>Returns the character that has the ASCII code value specified by the argument.</td>
</tr>
<tr>
<td>&quot;COLLATION_KEY_BIT&quot; on page 373</td>
<td>Returns a VARCHAR FOR BIT DATA string representing the collation key of the specified string-expression in the specified collation-name.</td>
</tr>
<tr>
<td>&quot;CONCAT&quot; on page 377</td>
<td>Returns a string that is the concatenation of two strings.</td>
</tr>
<tr>
<td>&quot;DECRYPT_BIN and DECRYPT_CHAR&quot; on page 403</td>
<td>Returns a value that is the result of decrypting encrypted data using a password string.</td>
</tr>
<tr>
<td>&quot;DIFFERENCE&quot; on page 407</td>
<td>Returns the difference between the sounds of the words in two argument strings, as determined by the SOUNDINDEX function. A value of 4 means the strings sound the same.</td>
</tr>
<tr>
<td>&quot;ENCRYPT&quot; on page 412</td>
<td>Returns a value that is the result of encrypting a data string expression.</td>
</tr>
<tr>
<td>&quot;GENERATE_UNIQUE&quot; on page 421</td>
<td>Returns a bit data character string that is unique compared to any other execution of the same function.</td>
</tr>
</tbody>
</table>
### Supported functions and administrative SQL routines and views

#### Table 40. String scalar functions (continued)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;GETHINT&quot; on page 423</td>
<td>Returns the password hint if one is found.</td>
</tr>
<tr>
<td>&quot;INITCAP&quot; on page 439</td>
<td>Returns a string with the first character of each word converted to uppercase and the rest to lowercase.</td>
</tr>
<tr>
<td>&quot;INSERT&quot; on page 441</td>
<td>Returns a string, where argument3 bytes have been deleted from argument1 (beginning at argument2), and argument4 has been inserted into argument1 (beginning at argument2).</td>
</tr>
<tr>
<td>&quot;INSTR&quot; on page 445</td>
<td>Returns the starting position of a string within another string.</td>
</tr>
<tr>
<td>&quot;INSTR&quot; on page 445</td>
<td>Returns a string in which all the SBCS characters have been converted to lowercase characters.</td>
</tr>
<tr>
<td>&quot;LCASE&quot; on page 450</td>
<td>Returns a string in which all the SBCS characters have been converted to lowercase characters using the rules from the Unicode standard associated with the specified locale.</td>
</tr>
<tr>
<td>&quot;LCASE (locale sensitive)&quot; on page 451</td>
<td>Returns a string in which all characters have been converted to lowercase characters using the rules from the Unicode standard associated with the specified locale.</td>
</tr>
<tr>
<td>&quot;LOWER (locale sensitive)&quot; on page 470</td>
<td>Returns a string in which all characters have been converted to lowercase characters using the rules from the Unicode standard associated with the specified locale.</td>
</tr>
<tr>
<td>&quot;LEFT&quot; on page 453</td>
<td>Returns the leftmost characters from a string.</td>
</tr>
<tr>
<td>&quot;LOCATE&quot; on page 459</td>
<td>Returns the starting position of one string within another string.</td>
</tr>
<tr>
<td>&quot;LOCATE_IN_STRING&quot; on page 463</td>
<td>Returns the starting position of the first occurrence of one string within another string.</td>
</tr>
<tr>
<td>&quot;LOWER&quot; on page 469</td>
<td>Returns a string in which all the characters have been converted to lowercase characters.</td>
</tr>
<tr>
<td>&quot;LPAD&quot; on page 472</td>
<td>Returns a string that is padded on the left with the specified character, or with blanks.</td>
</tr>
<tr>
<td>&quot;LTRIM&quot; on page 475</td>
<td>Removes blanks from the beginning of a string expression.</td>
</tr>
<tr>
<td>&quot;OCTET_LENGTH&quot; on page 494</td>
<td>Returns the length of an expression in octets (bytes).</td>
</tr>
<tr>
<td>&quot;OVERLAY&quot; on page 495</td>
<td>Returns a string in which, beginning at start in the specified source-string, length of the specified code units have been deleted and insert-string has been inserted.</td>
</tr>
<tr>
<td>&quot;POSITION&quot; on page 500</td>
<td>Returns the starting position of argument2 within argument1.</td>
</tr>
<tr>
<td>&quot;POSSTR&quot; on page 503</td>
<td>Returns the starting position of one string within another string.</td>
</tr>
<tr>
<td>&quot;REPEAT&quot; on page 518</td>
<td>Returns a character string composed of argument1 repeated argument2 times.</td>
</tr>
<tr>
<td>&quot;REPLACE&quot; on page 519</td>
<td>Replaces all occurrences of argument2 in argument1 with argument3.</td>
</tr>
<tr>
<td>&quot;RIGHT&quot; on page 523</td>
<td>Returns the rightmost characters from a string.</td>
</tr>
<tr>
<td>&quot;RPAD&quot; on page 534</td>
<td>Returns a string that is padded on the right with the specified character, string, or with blanks.</td>
</tr>
<tr>
<td>&quot;RTRIM&quot; on page 537</td>
<td>Removes blanks from the end of a string expression.</td>
</tr>
<tr>
<td>&quot;SOUNDEX&quot; on page 548</td>
<td>Returns a 4-character code representing the sound of the words in the argument. This result can be compared with the sound of other strings.</td>
</tr>
<tr>
<td>&quot;SPACE&quot; on page 549</td>
<td>Returns a character string that consists of a specified number of blanks.</td>
</tr>
</tbody>
</table>
### Supported functions and administrative SQL routines and views

#### Table 40. String scalar functions (continued)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;STRIP&quot; on page 551</td>
<td>Removes leading or trailing blanks or other specified leading or trailing characters from a string expression.</td>
</tr>
<tr>
<td>&quot;SUBSTR&quot; on page 552</td>
<td>Returns a substring of a string.</td>
</tr>
<tr>
<td>&quot;SUBSTRB&quot; on page 555</td>
<td>Returns a substring of a string.</td>
</tr>
<tr>
<td>&quot;SUBSTRING&quot; on page 558</td>
<td>Returns a substring of a string.</td>
</tr>
<tr>
<td>&quot;TRANSLATE&quot; on page 585</td>
<td>Returns a string in which one or more characters in a string are converted to other characters.</td>
</tr>
<tr>
<td>&quot;TRIM&quot; on page 588</td>
<td>Removes leading or trailing blanks or other specified leading or trailing characters from a string expression.</td>
</tr>
<tr>
<td>&quot;UCASE&quot; on page 599</td>
<td>The UCASE function is identical to the TRANSLATE function except that only the first argument (char-string-exp) is specified.</td>
</tr>
<tr>
<td>&quot;UCASE (locale sensitive)&quot; on page 600</td>
<td>Returns a string in which all characters have been converted to uppercase characters using the rules from the Unicode standard associated with the specified locale.</td>
</tr>
<tr>
<td>&quot;UPPER&quot; on page 601</td>
<td>Returns a string in which all the characters have been converted to uppercase characters.</td>
</tr>
<tr>
<td>&quot;UPPER (locale sensitive)&quot; on page 602</td>
<td>Returns a string in which all characters have been converted to uppercase characters using the rules from the Unicode standard associated with the specified locale.</td>
</tr>
</tbody>
</table>

#### Table 41. Table functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;BASE_TABLE&quot; on page 669</td>
<td>Returns both the object name and schema name of the object found after any alias chains have been resolved.</td>
</tr>
<tr>
<td>&quot;UNNEST&quot; on page 671</td>
<td>Returns a result table that includes a row for each element of the specified array.</td>
</tr>
<tr>
<td>&quot;XMLTABLE&quot; on page 673</td>
<td>Returns a table from the evaluation of XQuery expressions, possibly using specified input arguments as XQuery variables. Each sequence item in the result sequence of the row XQuery expression represents a row of the result table.</td>
</tr>
</tbody>
</table>

#### Table 42. XML functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;PARAMETER&quot; on page 499</td>
<td>Represents a position in an SQL statement where the value is provided dynamically by XQuery as part of the invocation of the db2-fn:sqlquery function.</td>
</tr>
<tr>
<td>&quot;XMLAGG&quot; on page 335</td>
<td>Returns an XML sequence containing an item for each non-null value in a set of XML values.</td>
</tr>
<tr>
<td>&quot;XMLATTRIBUTES&quot; on page 628</td>
<td>Constructs XML attributes from the arguments.</td>
</tr>
<tr>
<td>&quot;XMLCOMMENT&quot; on page 630</td>
<td>Returns an XML value with a single XQuery comment node with the input argument as the content.</td>
</tr>
<tr>
<td>&quot;XMLCONCAT&quot; on page 631</td>
<td>Returns a sequence containing the concatenation of a variable number of XML input arguments.</td>
</tr>
<tr>
<td>&quot;XMLDOCUMENT&quot; on page 632</td>
<td>Returns an XML value with a single XQuery document node with zero or more children nodes.</td>
</tr>
</tbody>
</table>
Table 42. XML functions (continued)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;XMLELEMENT&quot; on page 634</td>
<td>Returns an XML value that is an XML element node.</td>
</tr>
<tr>
<td>&quot;XMLFOREST&quot; on page 641</td>
<td>Returns an XML value that is a sequence of XML element nodes.</td>
</tr>
<tr>
<td>&quot;XMLGROUP&quot; on page 337</td>
<td>Returns an XML value with a single XQuery document node containing one top-level element node.</td>
</tr>
<tr>
<td>&quot;XMLNAMESPACES&quot; on page 644</td>
<td>Constructs namespace declarations from the arguments.</td>
</tr>
<tr>
<td>&quot;XMLPARSE&quot; on page 646</td>
<td>Parses the argument as an XML document and returns an XML value.</td>
</tr>
<tr>
<td>&quot;XMLPI&quot; on page 649</td>
<td>Returns an XML value with a single XQuery processing instruction node.</td>
</tr>
<tr>
<td>&quot;XMLQUERY&quot; on page 650</td>
<td>Returns an XML value from the evaluation of an XQuery expression possibly using specified input arguments as XQuery variables.</td>
</tr>
<tr>
<td>&quot;XMLROW&quot; on page 653</td>
<td>Returns an XML value with a single XQuery document node containing one top-level element node.</td>
</tr>
<tr>
<td>&quot;XMLSERIALIZE&quot; on page 655</td>
<td>Returns a serialized XML value of the specified data type generated from the argument.</td>
</tr>
<tr>
<td>&quot;XMLTABLE&quot; on page 673</td>
<td>Returns a table from the evaluation of XQuery expressions, possibly using specified input arguments as XQuery variables. Each sequence item in the result sequence of the row XQuery expression represents a row of the result table.</td>
</tr>
<tr>
<td>&quot;XMLTEXT&quot; on page 657</td>
<td>Returns an XML value with a single XQuery text node having the input argument as the content.</td>
</tr>
<tr>
<td>&quot;XMLVALIDATE&quot; on page 659</td>
<td>Returns a copy of the input XML value augmented with information obtained from XML schema validation, including default values.</td>
</tr>
<tr>
<td>&quot;XMLXSROBJECTID&quot; on page 663</td>
<td>Returns the XSR object identifier of the XML schema used to validate the XML document that is specified in the argument.</td>
</tr>
<tr>
<td>&quot;XSLTRANSFORM&quot; on page 664</td>
<td>Converts XML data into other formats, including the conversion of XML documents that conform to one XML schema into documents that conform to another schema.</td>
</tr>
</tbody>
</table>

Aggregate functions

The argument of an aggregate function is a set of values derived from an expression. The expression can include columns, but cannot include a `scalar-fullselect`, another aggregate function, or an XMLQUERY or XMLEXISTS expression (SQLSTATE 42607). The scope of the set is a group or an intermediate result table.

If a GROUP BY clause is specified in a query, and the intermediate result of the FROM, WHERE, GROUP BY, and HAVING clauses is the empty set, the aggregate functions are not applied; the result of the query is the empty set; the SQLCODE is set to +100; and the SQLSTATE is set to '02000'.
Aggregate functions

If a GROUP BY clause is not specified in a query, and the intermediate result of the FROM, WHERE, and HAVING clauses is the empty set, the aggregate functions are applied to the empty set.

For example, the result of the following SELECT statement is the number of distinct values of JOBCODE for employees in department D01:

```
SELECT COUNT(DISTINCT JOBCODE)
FROM CORPODATA.EMPLOYEE
WHERE WORKDEPT = 'D01'
```

The keyword DISTINCT is not considered to be an argument of the function, but rather a specification of an operation that is performed before the function is applied. If DISTINCT is specified, duplicate values are eliminated. When interpreting the DISTINCT clause for decimal floating-point values that are numerically equal, the number of significant digits in the value is not considered. For example, the decimal floating-point number 123.00 is not distinct from the decimal floating-point number 123. The representation of the number returned from the query will be any one of the representations encountered (for example, either 123.00 or 123).

If ALL is implicitly or explicitly specified, duplicate values are not eliminated.

For compatibility with other SQL implementations, UNIQUE can be specified as a synonym for DISTINCT in aggregate functions.

Expressions can be used in aggregate functions. For example:

```
SELECT MAX(BONUS + 1000)
INTO :TOP_SALESREP_BONUS
FROM EMPLOYEE
WHERE COMM > 5000
```

Aggregate functions can be qualified with a schema name (for example, SYSIBM.COUNT(*)).
The schema is SYSIBM.

The ARRAY_AGG function aggregates a set of elements into an array. The data type of the expression must be a data type that can be specified in a CREATE TYPE (array) statement (SQLSTATE 429C2).

If sort-key is specified, it determines the order of the aggregated elements in the array. If sort-key is not specified, the ordering of elements within the array is not deterministic. If sort-key is not specified, and ARRAY_AGG is specified more than once in the same SELECT clause, the same ordering of elements within the array is used for each result of ARRAY_AGG.

If a SELECT clause has multiple occurrences of XMLAGG or ARRAY_AGG that specify sort-key, all the sort keys must be identical (SQLSTATE 428GZ).

The ARRAY_AGG function can only be specified within an SQL procedure in the following specific contexts (SQLSTATE 42887):

- The select-list of a SELECT INTO statement
- The select-list of a fullselect in the definition of a cursor that is not scrollable
- The select-list of a scalar subquery on the right side of a SET statement

ARRAY_AGG cannot be used as part of an OLAP function (SQLSTATE 42887). The SELECT statement that uses ARRAY_AGG cannot contain an ORDER BY clause or a DISTINCT clause, and the SELECT clause or HAVING clause cannot contain a subquery or call an SQL function (SQLSTATE 42887).

ARRAY_AGG cannot be used to produce an associative array or an array with a row element data type (SQLSTATE 42846).

Example:

- Given the following DDL:
  
  ```sql
  CREATE TYPE PHONELIST AS DECIMAL(10, 0)ARRAY[10]
  CREATE TABLE EMPLOYEE (ID INTEGER NOT NULL,
  PRIORITY INTEGER NOT NULL,
  PHONENUMBER DECIMAL(10, 0),
  PRIMARY KEY(ID, PRIORITY))
  ```

  Create a procedure that uses a SELECT INTO statement to return the prioritized list of contact numbers under which an employee can be reached.

  ```sql
  CREATE PROCEDURE GETPHONENUMBERS (IN EMPID INTEGER, 
  OUT NUMBERS PHONELIST)
  BEGIN 
  SELECT ARRAY_AGG(PHONENUMBER ORDER BY PRIORITY)
  ```
Create a procedure that uses a SET statement to return the list of an employee’s contact numbers in an arbitrary order.

```sql
CREATE PROCEDURE GETPHONENUMBERS
  (IN EMPID INTEGER,
   OUT NUMBERS PHONELIST)
BEGIN
  SET NUMBERS =
    (SELECT ARRAY_AGG(PHONENUMBER)
    FROM EMPLOYEE
    WHERE ID = EMPID);
END
```
The AVG function returns the average of a set of numbers.

The argument values must be numbers (built-in types only) and their sum must be within the range of the data type of the result, except for a decimal result data type. For decimal results, their sum must be within the range supported by a decimal data type having a precision of 31 and a scale identical to the scale of the argument values. The result can be null.

The data type of the result is the same as the data type of the argument values, except that:

- The result is a large integer if the argument values are small integers.
- The result is double-precision floating point if the argument values are single-precision floating point.
- The result is DECFLOAT(34) if the argument is DECFLOAT(n).

If the data type of the argument values is decimal with precision \( p \) and scale \( s \), the precision of the result is 31 and the scale is \( 31-p+s \).

The function is applied to the set of values derived from the argument values by the elimination of null values. If DISTINCT is specified, redundant duplicate values are eliminated. When interpreting the DISTINCT clause for decimal floating-point values that are numerically equal, the number of significant digits in the value is not considered. For example, the decimal floating-point number 123.00 is not distinct from the decimal floating-point number 123. The representation of the number returned from the query will be any one of the representations encountered (for example, either 123.00 or 123).

If the function is applied to an empty set, the result is a null value. Otherwise, the result is the average value of the set.

The order in which the values are added is undefined, but every intermediate result must be within the range of the result data type.

If the type of the result is integer, the fractional part of the average is lost.

Examples:

- Using the PROJECT table, set the host variable AVERAGE (decimal(5,2)) to the average staffing level (PRSTAFF) of projects in department (DEPTNO) 'D11'.

```
SELECT AVG(PRSTAFF)
INTO :AVERAGE
FROM PROJECT
WHERE DEPTNO = 'D11'
```

Results in AVERAGE being set to 4.25 (that is 17/4) when using the sample table.
- Using the PROJECT table, set the host variable ANY_CALC (decimal(5,2)) to the average of each unique staffing level value (PRSTAFF) of projects in department (DEPTNO) 'D11'.

```
SELECT AVG(DISTINCT PRSTAFF)
INTO :ANY_CALC
FROM PROJECT
WHERE DEPTNO = 'D11'
```

Results in ANY_CALC being set to 4.66 (that is 14/3) when using the sample table.
The schema is SYSIBM.

The CORRELATION function returns the coefficient of correlation of a set of number pairs.

The argument values must be numbers.

If either argument is decimal floating-point, the result is DECIMAL(34); otherwise, the result is a double-precision floating-point number. The result can be null. When not null, the result is between -1 and 1.

The function is applied to the set of \((\text{expression}_1, \text{expression}_2)\) pairs derived from the argument values by the elimination of all pairs for which either \(\text{expression}_1\) or \(\text{expression}_2\) is null.

If the function is applied to an empty set, or if either STDDEV(\(\text{expression}_1\)) or STDDEV(\(\text{expression}_2\)) is equal to zero, the result is a null value. Otherwise, the result is the correlation coefficient for the value pairs in the set. The result is equivalent to the following expression:

\[
\frac{\text{COVARIANCE}(\text{expression}_1, \text{expression}_2)}{(\text{STDDEV}(\text{expression}_1) \times \text{STDDEV}(\text{expression}_2))}
\]

The order in which the values are aggregated is undefined, but every intermediate result must be within the range of the result data type.

CORR can be specified in place of CORRELATION.

Example:

- Using the EMPLOYEE table, set the host variable CORRLN (double-precision floating point) to the correlation between salary and bonus for those employees in department (WORKDEPT) 'A00'.

```sql
SELECT CORRELATION(SALARY, BONUS) INTO :CORRLN
FROM EMPLOYEE
WHERE WORKDEPT = 'A00'
```

CORRLN is set to approximately 9.99853953399538E-001 when using the sample table.
The schema is SYSIBM.

The COUNT function returns the number of rows or values in a set of rows or values.

If DISTINCT is specified, the resulting data type of expression cannot be a BLOB, CLOB, DBCLOB, XML, distinct type on any of these types, or structured type (SQLSTATE 42907). Otherwise the result data type of expression can be any data type.

The result of the function is a large integer. The result cannot be null.

The argument of COUNT(*) is a set of rows. The result is the number of rows in the set. A row that includes only NULL values is included in the count.

The argument of COUNT(DISTINCT expression) is a set of values. The function is applied to the set of values derived from the argument values by the elimination of null and duplicate values. The result is the number of different non-null values in the set.

The argument of COUNT(expression) or COUNT(ALL expression) is a set of values. The function is applied to the set of values derived from the argument values by the elimination of null values. The result is the number of non-null values in the set, including duplicates.

Examples:
- Using the EMPLOYEE table, set the host variable FEMALE (int) to the number of rows where the value of the SEX column is 'F'.

```
SELECT COUNT(*)
INTO :FEMALE
FROM EMPLOYEE
WHERE SEX = 'F'
```

Results in FEMALE being set to 13 when using the sample table.

- Using the EMPLOYEE table, set the host variable FEMALE_IN_DEPT (int) to the number of departments (WORKDEPT) that have at least one female as a member.

```
SELECT COUNT(DISTINCT WORKDEPT)
INTO :FEMALE_IN_DEPT
FROM EMPLOYEE
WHERE SEX = 'F'
```

Results in FEMALE_IN_DEPT being set to 5 when using the sample table.

(There is at least one female in departments A00, C01, D11, D21, and E11.)
The schema is SYSIBM.

The COUNT_BIG function returns the number of rows or values in a set of rows or values. It is similar to COUNT except that the result can be greater than the maximum value of integer.

If DISTINCT is specified, the resulting data type of expression cannot be a BLOB, CLOB, DBCLOB, XML, distinct type on any of these types, or structured type (SQLSTATE 42907). Otherwise the result data type of expression can be any data type.

The result of the function is a decimal with precision 31 and scale 0. The result cannot be null.

The argument of COUNT_BIG(*) is a set of rows. The result is the number of rows in the set. A row that includes only NULL values is included in the count.

The argument of COUNT_BIG(DISTINCT expression) is a set of values. The function is applied to the set of values derived from the argument values by the elimination of null and duplicate values. The result is the number of different non-null values in the set.

The argument of COUNT_BIG(expression) or COUNT_BIG(ALL expression) is a set of values. The function is applied to the set of values derived from the argument values by the elimination of null values. The result is the number of non-null values in the set, including duplicates.

Examples:
- Refer to COUNT examples and substitute COUNT_BIG for occurrences of COUNT. The results are the same except for the data type of the result.
- Some applications may require the use of COUNT but need to support values larger than the largest integer. This can be achieved by use of sourced user-defined functions and setting the SQL path. The following series of statements shows how to create a sourced function to support COUNT(*) based on COUNT_BIG and returning a decimal value with a precision of 15. The SQL path is set such that the sourced function based on COUNT_BIG is used in subsequent statements such as the query shown.

```sql
CREATE FUNCTION RICK.COUNT() RETURNS DECIMAL(15,0)
SOURCE SYSIBM.COUNT_BIG();
SET CURRENT PATH RICK, SYSTEM PATH;
SELECT COUNT(*) FROM EMPLOYEE;
```

Note how the sourced function is defined with no parameters to support COUNT(*). This only works if you name the function COUNT and do not qualify the function with the schema name when it is used. To get the same effect as COUNT(*) with a name other than COUNT, invoke the function with no parameters. Thus, if RICK.COUNT had been defined as RICK.MYCOUNT instead, the query would have to be written as follows:
SELECT MYCOUNT() FROM EMPLOYEE;

If the count is taken on a specific column, the sourced function must specify the type of the column. The following statements create a sourced function that will take any CHAR column as an argument and use COUNT_BIG to perform the counting.

CREATE FUNCTION RICK.COUNT(CHAR()) RETURNS DOUBLE
SOURCE SYSIBM.COUNT_BIG(CHAR());
SELECT COUNT(DISTINCT WORKDEPT) FROM EMPLOYEE;

COUNT_BIG
The schema is SYSIBM.

The COVARIANCE function returns the (population) covariance of a set of number pairs.

The argument values must be numbers.

If either argument is decimal floating-point, the result is DECFLOAT(34); otherwise, the result is a double-precision floating-point number. The result can be null.

The function is applied to the set of \((\text{expression1}, \text{expression2})\) pairs derived from the argument values by the elimination of all pairs for which either \text{expression1} or \text{expression2} is null.

If the function is applied to an empty set, the result is a null value. Otherwise, the result is the covariance of the value pairs in the set. The result is equivalent to the following:

1. Let avgexp1 be the result of AVG(\text{expression1}) and let avgexp2 be the result of AVG(\text{expression2}).
2. The result of COVARIANCE(\text{expression1}, \text{expression2}) is AVG( (\text{expression1} - avgexp1) * (\text{expression2} - avgexp2 )

The order in which the values are aggregated is undefined, but every intermediate result must be within the range of the result data type.

COVAR can be specified in place of COVARIANCE.

Example:

- Using the EMPLOYEE table, set the host variable COVARNCE (double-precision floating point) to the covariance between salary and bonus for those employees in department (WORKDEPT) ‘A00’.

  ```sql
  SELECT COVARIANCE(SALARY, BONUS)
  INTO :COVARNCE
  FROM EMPLOYEE
  WHERE WORKDEPT = 'A00'
  ```

COVARNCE is set to approximately 1.6888888888889E+006 when using the sample table.
GROUPING

The schema is SYSIBM.

Used in conjunction with grouping-sets and super-groups, the GROUPING function returns a value that indicates whether or not a row returned in a GROUP BY answer set is a row generated by a grouping set that excludes the column represented by expression.

The argument can be of any type, but must be an item of a GROUP BY clause.

The result of the function is a small integer. It is set to one of the following values:

1  The value of expression in the returned row is a null value, and the row was generated by the super-group. This generated row can be used to provide sub-total values for the GROUP BY expression.
0  The value is other than the above.

Example:

The following query:

```sql
SELECT SALES_DATE, SALES_PERSON,
       SUM(SALES) AS UNITS_SOLD,
       GROUPING(SALES_DATE) AS DATE_GROUP,
       GROUPING(SALES_PERSON) AS SALES_GROUP
FROM SALES
GROUP BY CUBE (SALES_DATE, SALES_PERSON)
ORDER BY SALES_DATE, SALES_PERSON
```

results in:

<table>
<thead>
<tr>
<th>SALES_DATE</th>
<th>SALES_PERSON</th>
<th>UNITS_SOLD</th>
<th>DATE_GROUP</th>
<th>SALES_GROUP</th>
</tr>
</thead>
<tbody>
<tr>
<td>12/31/1995</td>
<td>GOUNOT</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>12/31/1995</td>
<td>LEE</td>
<td>6</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>12/31/1995</td>
<td>LUCHESSI</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>12/31/1995</td>
<td>-</td>
<td>8</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>03/29/1996</td>
<td>GOUNOT</td>
<td>11</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>03/29/1996</td>
<td>LEE</td>
<td>12</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>03/29/1996</td>
<td>LUCHESSI</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>03/29/1996</td>
<td>-</td>
<td>27</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>03/30/1996</td>
<td>GOUNOT</td>
<td>21</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>03/30/1996</td>
<td>LEE</td>
<td>21</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>03/30/1996</td>
<td>LUCHESSI</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>03/30/1996</td>
<td>-</td>
<td>46</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>03/31/1996</td>
<td>GOUNOT</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>03/31/1996</td>
<td>LEE</td>
<td>27</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>03/31/1996</td>
<td>LUCHESSI</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>03/31/1996</td>
<td>-</td>
<td>31</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>04/01/1996</td>
<td>GOUNOT</td>
<td>14</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>04/01/1996</td>
<td>LEE</td>
<td>25</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>04/01/1996</td>
<td>LUCHESSI</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>04/01/1996</td>
<td>-</td>
<td>43</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>-</td>
<td>GOUNOT</td>
<td>50</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>-</td>
<td>LEE</td>
<td>91</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>-</td>
<td>LUCHESSI</td>
<td>14</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>155</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
An application can recognize a SALES_DATE sub-total row by the fact that the value of DATE_GROUP is 0 and the value of SALES_GROUP is 1. A SALES_PERSON sub-total row can be recognized by the fact that the value of DATE_GROUP is 1 and the value of SALES_GROUP is 0. A grand total row can be recognized by the value 1 for both DATE_GROUP and SALES_GROUP.
The schema is SYSIBM.

The MAX function returns the maximum value in a set of values.

The argument values can be of any built-in type other than a LOB string.

The resulting data type of expression cannot be a BLOB, CLOB, DBCLOB, distinct type on any of these types, or structured type (SQLSTATE 42907).

The data type, length and code page of the result are the same as the data type, length and code page of the argument values. The result is considered to be a derived value and can be null.

The function is applied to the set of values derived from the argument values by the elimination of null values.

If the function is applied to an empty set, the result is a null value. Otherwise, the result is the maximum value in the set.

The specification of DISTINCT has no effect on the result and therefore is not recommended. It is included for compatibility with other relational systems.

Examples:

- Using the EMPLOYEE table, set the host variable MAX_SALARY (decimal(7,2)) to the maximum monthly salary (SALARY/12) value.

```
SELECT MAX(SALARY) / 12 INTO :MAX_SALARY FROM EMPLOYEE
```

Results in MAX_SALARY being set to 4395.83 when using the sample table.

- Using the PROJECT table, set the host variable LAST_PROJ(char(24)) to the project name (PROJNAME) that comes last in the collating sequence.

```
SELECT MAX(PROJNAME) INTO :LAST_PROJ FROM PROJECT
```

Results in LAST_PROJ being set to ‘WELD LINE PLANNING’ when using the sample table.

- Similar to the previous example, set the host variable LAST_PROJ (char(40)) to the project name that comes last in the collating sequence when a project name is concatenated with the host variable PROJSUPP. PROJSUPP is ‘_Support’; it has a char(8) data type.

```
SELECT MAX(PROJNAME CONCAT PROJSUPP) INTO :LAST_PROJ FROM PROJECT
```

Results in LAST_PROJ being set to ‘WELD LINE PLANNING_SUPPORT’ when using the sample table.
The MIN function returns the minimum value in a set of values.

The argument values can be of any built-in type other than a LOB string.

The resulting data type of expression cannot be a BLOB, CLOB, DBCLOB, distinct type on any of these types, or structured type (SQLSTATE 42907).

The data type, length, and code page of the result are the same as the data type, length, and code page of the argument values. The result is considered to be a derived value and can be null.

The function is applied to the set of values derived from the argument values by the elimination of null values.

If this function is applied to an empty set, the result of the function is a null value. Otherwise, the result is the minimum value in the set.

The specification of DISTINCT has no effect on the result and therefore is not recommended. It is included for compatibility with other relational systems.

Examples:

- Using the EMPLOYEE table, set the host variable COMM_SPREAD (decimal(7,2)) to the difference between the maximum and minimum commission (COMM) for the members of department (WORKDEPT) 'D11'.

  ```sql
  SELECT MAX(COMM) - MIN(COMM)
  INTO :COMM_SPREAD
  FROM EMPLOYEE
  WHERE WORKDEPT = 'D11'
  ```

  Results in COMM_SPREAD being set to 1118 (that is, 2580 - 1462) when using the sample table.

- Using the PROJECT table, set the host variable (FIRST_FINISHED (char(10)) to the estimated ending date (PRENDATE) of the first project scheduled to be completed.

  ```sql
  SELECT MIN(PRENDATE)
  INTO :FIRST_FINISHED
  FROM PROJECT
  ```

  Results in FIRST_FINISHED being set to '1982-09-15' when using the sample table.
The schema is SYSIBM.

The regression functions support the fitting of an ordinary-least-squares regression line of the form \( y = a \cdot x + b \) to a set of number pairs. The first element of each pair (\( \text{expression1} \)) is interpreted as a value of the dependent variable (that is, a "y value"). The second element of each pair (\( \text{expression2} \)) is interpreted as a value of the independent variable (that is, an "x value").

The REGR_COUNT function returns the number of non-null number pairs used to fit the regression line (see below).

The REGR_INTERCEPT (or REGR_ICPT) function returns the y-intercept of the regression line ("b" in the above equation).

The REGR_R2 function returns the coefficient of determination ("R-squared" or "goodness-of-fit") for the regression.

The REGR_SLOPE function returns the slope of the line ("a" in the above equation).

The \( \text{REGR_AVGX, REGR_AVGY, REGR_SXX, REGR_SXY, and REGR_SYY} \) functions return quantities that can be used to compute various diagnostic statistics needed for the evaluation of the quality and statistical validity of the regression model (see below).

The argument values must be numbers.

The data type of the result of REGR_COUNT is integer. For the remaining functions, if either argument is DECIMAL\( (n) \), the data type of the result is DECIMAL\( (34) \); otherwise, the data type of the result is double-precision floating-point. If either argument is a special decimal floating-point value, the rules for general arithmetic operations for decimal floating-point apply. See "General arithmetic operation rules for decimal floating-point" in "General arithmetic operation rules for decimal floating-point" on page 223.

The result can be null. When not null, the result of REGR_R2 is between 0 and 1, and the result of both REGR_SXX and REGR_SYY is non-negative.

Each function is applied to the set of (\( \text{expression1}, \text{expression2} \)) pairs derived from the argument values by the elimination of all pairs for which either \( \text{expression1} \) or \( \text{expression2} \) is null.
Regression functions

If the set is not empty and \( \text{VARIANCE(expression2)} \) is positive, \( \text{REGR_COUNT} \) returns the number of non-null pairs in the set, and the remaining functions return results that are defined as follows:

\[
\text{REGR\_SLOPE(expression1, expression2)} = \frac{\text{COVARIANCE(expression1, expression2)}}{\text{VARIANCE(expression2)}}
\]

\[
\text{REGR\_INTERCEPT(expression1, expression2)} = \text{AVG(expression1)} - \text{REGR\_SLOPE(expression1, expression2)} \times \text{AVG(expression2)}
\]

\[
\text{REGR\_R2(expression1, expression2)} = \begin{cases} 
\text{POWER(CORRELATION(expression1, expression2), 2)} & \text{if } \text{VARIANCE(expression1)}>0 \\
1 & \text{if } \text{VARIANCE(expression1)}=0 
\end{cases}
\]

\[
\text{REGR\_AVGX(expression1, expression2)} = \text{AVG(expression2)}
\]

\[
\text{REGR\_AVGY(expression1, expression2)} = \text{AVG(expression1)}
\]

\[
\text{REGR\_SXX(expression1, expression2)} = \text{REGR\_COUNT(expression1, expression2)} \times \text{VARIANCE(expression2)}
\]

\[
\text{REGR\_SYY(expression1, expression2)} = \text{REGR\_COUNT(expression1, expression2)} \times \text{VARIANCE(expression1)}
\]

\[
\text{REGR\_SXY(expression1, expression2)} = \text{REGR\_COUNT(expression1, expression2)} \times \text{COVARIANCE(expression1, expression2)}
\]

If the set is not empty and \( \text{VARIANCE(expression2)} \) is equal to zero, then the regression line either has infinite slope or is undefined. In this case, the functions \( \text{REGR\_SLOPE}, \text{REGR\_INTERCEPT}, \) and \( \text{REGR\_R2} \) each return a null value, and the remaining functions return values as defined above. If the set is empty, \( \text{REGR\_COUNT} \) returns zero and the remaining functions return a null value.

The order in which the values are aggregated is undefined, but every intermediate result must be within the range of the result data type.

The regression functions are all computed simultaneously during a single pass through the data. In general, it is more efficient to use the regression functions to compute the statistics needed for a regression analysis than to perform the equivalent computations using ordinary column functions such as \( \text{AVERAGE}, \text{VARIANCE}, \text{COVARIANCE} \), and so forth.

The usual diagnostic statistics that accompany a linear-regression analysis can be computed in terms of the above functions. For example:

**Adjusted R2**

\[
1 - (1 - \text{REGR\_R2}) \times ((\text{REGR\_COUNT} - 1) / (\text{REGR\_COUNT} - 2))
\]

**Standard error**

\[
\sqrt{((\text{REGR\_SYY}-(\text{POWER(\text{REGR\_SXY},2)/\text{REGR\_SXX})))/ (\text{REGR\_COUNT}-2))}
\]

**Total sum of squares**

\[
\text{REGR\_SYY}
\]

**Regression sum of squares**

\[
\text{POWER(\text{REGR\_SXY},2)/\text{REGR\_SXX}}
\]

**Residual sum of squares**

\[
(\text{Total sum of squares})-(\text{Regression sum of squares})
\]

**t statistic for slope**

\[
\text{REGR\_SLOPE} \times \sqrt{\text{REGR\_SXX}} / (\text{Standard error})
\]

**t statistic for y-intercept**

\[
\text{REGR\_INTERCEPT} / ((\text{Standard error}) \times \sqrt{1/(\text{REGR\_COUNT})+(\text{POWER(\text{REGR\_AVGX},2)/\text{REGR\_SXX}})})
\]
Example:

- Using the EMPLOYEE table, compute an ordinary-least-squares regression line that expresses the bonus of an employee in department (WORKDEPT) 'A00' as a linear function of the employee's salary. Set the host variables SLOPE, ICPT, RSQR (double-precision floating point) to the slope, intercept, and coefficient of determination of the regression line, respectively. Also set the host variables AVGSAL and AVGBONUS to the average salary and average bonus, respectively, of the employees in department 'A00', and set the host variable CNT (integer) to the number of employees in department 'A00' for whom both salary and bonus data are available. Store the remaining regression statistics in host variables SXX, SYY, and SXY.

```
SELECT REGR_SLOPE(BONUS, SALARY), REGR_INTERCEPT(BONUS, SALARY),
       REGR_R2(BONUS, SALARY), REGR_COUNT(BONUS, SALARY),
       REGR_AVGX(BONUS, SALARY), REGR_AVGY(BONUS, SALARY),
       REGR_SXX(BONUS, SALARY), REGR_SYY(BONUS, SALARY),
       REGR_SXY(BONUS, SALARY)
INTO :SLOPE, :ICPT,
       :RSQR, :CNT,
       :AVGSAL, :AVGBONUS,
       :SXX, :SYY,
       :SXY
FROM EMPLOYEE
WHERE WORKDEPT = 'A00'
```

When using the sample table, the host variables are set to the following approximate values:

- SLOPE: +1.71002671916749E-002
- ICPT: +1.00871888623260E+002
- RSQR: +9.99707928128685E-001
- CNT: 3
- AVGSAL: +4.28333333333333E+004
- AVGBONUS: +8.33333333333333E+002
- SXX: +2.96291666666667E+008
- SYY: +8.66666666666667E+004
- SXY: +5.06666666666667E+006
The schema is SYSIBM.

The STDDEV function returns the standard deviation (/n) of a set of numbers. The formula used to calculate STDDEV is:

\[
STDDEV = \sqrt{VARIANCE}
\]

where \(\sqrt{VARIANCE}\) is the square root of the variance.

The argument values must be numbers.

If the argument is DECFLOAT(n), the result is DECFLOAT(n); otherwise, the result is double-precision floating-point. The result can be null.

The function is applied to the set of values derived from the argument values by the elimination of null values. If DISTINCT is specified, redundant duplicate values are eliminated. When interpreting the DISTINCT clause for decimal floating-point values that are numerically equal, the number of significant digits in the value is not considered. For example, the decimal floating-point number 123.00 is not distinct from the decimal floating-point number 123. The representation of the number returned from the query will be any one of the representations encountered (for example, either 123.00 or 123).

If the function is applied to an empty set, the result is a null value. Otherwise, the result is the standard deviation of the values in the set.

The order in which the values are aggregated is undefined, but every intermediate result must be within the range of the result data type.

Example:

- Using the EMPLOYEE table, set the host variable DEV (double-precision floating point) to the standard deviation of the salaries of employees in department (WORKDEPT) 'A00'.

  ```sql
  SELECT STDDEV(SALARY)
  INTO :DEV
  FROM EMPLOYEE
  WHERE WORKDEPT = 'A00'
  ```

DEV is set to a number with an approximate value of 9938.00.
The schema is SYSIBM.

The SUM function returns the sum of a set of numbers.

The argument values must be numbers (built-in types only) and their sum must be within the range of the data type of the result.

The data type of the result is the same as the data type of the argument values, except that:

- The result is a large integer if the argument values are small integers.
- The result is double-precision floating point if the argument values are single-precision floating point.
- The result is DECFLOAT(34) if the argument is DECFLOAT(n).

If the data type of the argument values is decimal, the precision of the result is 31 and the scale is the same as the scale of the argument values. The result can be null.

The function is applied to the set of values derived from the argument values by the elimination of null values. If DISTINCT is specified, redundant duplicate values are also eliminated. When interpreting the DISTINCT clause for decimal floating-point values that are numerically equal, the number of significant digits in the value is not considered. For example, the decimal floating-point number 123.00 is not distinct from the decimal floating-point number 123. The representation of the number returned from the query will be any one of the representations encountered (for example, either 123.00 or 123).

If the function is applied to an empty set, the result is a null value. Otherwise, the result is the sum of the values in the set.

The order in which the values are aggregated is undefined, but every intermediate result must be within the range of the result data type.

Example:

- Using the EMPLOYEE table, set the host variable JOB_BONUS (decimal(9,2)) to the total bonus (BONUS) paid to clerks (JOB='CLERK').

```sql
SELECT SUM(BONUS) INTO :JOB_BONUS
FROM EMPLOYEE
WHERE JOB = 'CLERK'
```

Results in JOB_BONUS being set to 2800 when using the sample table.
The schema is SYSIBM.

The VARIANCE function returns the variance of a set of numbers.

The argument values must be numbers.

If the argument is DECFLOAT(n), the result is DECFLOAT(n); otherwise, the result is double-precision floating-point. The result can be null.

The function is applied to the set of values derived from the argument values by the elimination of null values. If DISTINCT is specified, redundant duplicate values are eliminated. When interpreting the DISTINCT clause for decimal floating-point values that are numerically equal, the number of significant digits in the value is not considered. For example, the decimal floating-point number 123.00 is not distinct from the decimal floating-point number 123. The representation of the number returned from the query will be any one of the representations encountered (for example, either 123.00 or 123).

If the function is applied to an empty set, the result is a null value. Otherwise, the result is the variance of the values in the set.

The order in which the values are added is undefined, but every intermediate result must be within the range of the result data type.

VAR can be specified in place of VARIANCE.

Example:

- Using the EMPLOYEE table, set the host variable VARNCE (double-precision floating point) to the variance of the salaries for those employees in department (WORKDEPT) 'A00'.

  `SELECT VARIANCE(SALARY) INTO :VARNCE FROM EMPLOYEE WHERE WORKDEPT = 'A00';`

Results in VARNCE being set to approximately 98763888.88 when using the sample table.
XMLAGG

The schema is SYSIBM. The function name cannot be specified as a qualified name.

The XMLAGG function returns an XML sequence containing an item for each non-null value in a set of XML values.

XML-expression
   Specifies an expression of data type XML.

ORDER BY
   Specifies the order of the rows from the same grouping set that are processed in the aggregation. If the ORDER BY clause is omitted, or if the ORDER BY clause cannot distinguish the order of the column data, the rows in the same grouping set are arbitrarily ordered.

sort-key
   The sort key can be a column name or a sort-key-expression. Note that if the sort key is a constant, it does not refer to the position of the output column (as in the ordinary ORDER BY clause), but it is simply a constant, which implies no sort key.

The data type of the result is XML.

The function is applied to the set of values derived from the argument values by the elimination of null values.

If the XML-expression argument can be null, the result can be null. If the set of values is empty, the result is the null value. Otherwise, the result is an XML sequence containing an item for each value in the set.

Note:
1. Support in OLAP expressions: XMLAGG cannot be used as a column function of an OLAP aggregation function (SQLSTATE 42601).

Example:

Note: XMLAGG does not insert blank spaces or new line characters in the output.
All example output has been formatted to enhance readability.
- Construct a department element for each department, containing a list of employees sorted by last name.

```sql
SELECT XMLSERIALIZE(
    CONTENT XMLELEMENT(
        NAME "Department", XMLATTRIBUTES(
            E.WORKDEPT AS "name"
        ),
        XMLAGG(
            XMLELEMENT(
                NAME "emp", E.LASTNAME
            )
        )
    ) ORDER BY E.LASTNAME
)`
XMLAGG

```sql
AS

AS "dept_list"
FROM EMPLOYEE E
WHERE E.WORKDEPT IN ('C01','E21')
GROUP BY WORKDEPT

This query produces the following result:

department_list
-----------------------
<Department name="C01">
  <emp>KWAN</emp>
  <emp>NICHOLLS</emp>
  <emp>QUINTANA</emp>
</Department>
<Department name="E21">
  <emp>GOUNOT</emp>
  <emp>LEE</emp>
  <emp>MEHTA</emp>
  <emp>SPENSER</emp>
</Department>
```
XMLGROUP

The XMLGROUP function returns an XML value with a single XQuery document node containing one top-level element node. This is an aggregate expression that will return a single-rooted XML document from a group of rows where each row is mapped to a row subelement.

Notes:
1. The same clause must not be specified more than once.

The schema is SYSIBM. The function name cannot be specified as a qualified name.

expression
The content of each generated XML element node (or the value of each generated attribute) is specified by an expression. The data type of expression cannot be a structured type (SQLSTATE 42884). The expression can be any SQL expression. If the expression is not a simple column reference, a qname-identifier must be specified.

AS qname-identifier
Specifies the XML element name or attribute name as an SQL identifier. The qname-identifier must be of the form of an XML qualified name, or QName (SQLSTATE 42634). See the W3C XML namespace specifications for more details on valid names. If the name is qualified, the namespace prefix must be declared within the scope (SQLSTATE 42635). If qname-identifier is not specified, expression must be a column name (SQLSTATE 42703). The element name or attribute name is created from the column name using the fully escaped mapping from a column name to an QName.

OPTION
Specifies additional options for constructing the XML value. If no OPTION clause is specified, the default behavior applies.

ROW row-name
Specifies the name of the element to which each row is mapped. If this option is not specified, the default element name is "row".
XMLGROUP

ROOT root-name
Specifies the name of the root element node. If this option is not specified, the default root element name is "rowset".

AS ATTRIBUTES
Specifies that each expression is mapped to an attribute value with column name or qname-identifier serving as the attribute name.

ORDER BY
Specifies the order of the rows from the same grouping set that are processed in the aggregation. If the ORDER BY clause is omitted, or if the ORDER BY clause cannot distinguish the order of the column data, the rows in the same grouping set are arbitrarily ordered.

sort-key
The sort key can be a column name or a sort-key-expression. Note that if the sort key is a constant, it does not refer to the position of the output column (as in the ordinary ORDER BY clause), but it is simply a constant, which implies no sort key.

Notes
The default behavior defines a simple mapping between a result set and an XML value. Some additional notes about function behavior apply:

- By default, each row is transformed into an XML element named "row" and each column is transformed into a nested element with the column name serving as the element name.
- The null handling behavior is NULL ON NULL. A NULL value in a column maps to the absence of the subelement. If all column values are NULL, no row element will be generated.
- The binary encoding scheme for BLOB and FOR BIT DATA data types is base64Binary encoding.
- By default, the elements corresponding to the rows in a group are children of a root element named "rowset".
- The order of the row subelements in the root element will be the same as the order in which the rows are returned in the query result set.
- A document node will be added implicitly to the root element to make the XML result a well-formed single-rooted XML document.

Examples
Assume the following table T1 with integer columns C1 and C2 that contain numeric data stored in a relational format.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>-</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

4 record(s) selected.

- The following example shows an XMLGroup query and output fragment with default behavior, using a single top-level element to represent the table:

```sql
SELECT XMLGROUP(C1, C2) FROM T1
```

```
<rowset>
  <row>
    <C1>1</C1>
  </row>
</rowset>
```
The following example shows an XMLGroup query and output fragment with attribute centric mapping. Instead of appearing as nested elements as in the previous example, relational data is mapped to element attributes:

```xml
SELECT XMLGROUP(C1, C2 OPTION AS ATTRIBUTES) FROM T1
```

```xml
<rowset>
  <row C1="1" C2="2"/>
  <row C2="2"/>
  <row C1="1"/>
</rowset>
```

1 record(s) selected.

The following example shows an XMLGroup query and output fragment with the default `<rowset>` root element replaced by `<document>` and the default `<row>` element replaced by `<entry>`. Columns C1 and C2 are returned as `<column1>` and `<column2>` elements, and the return set is ordered by column C1:

```xml
SELECT XMLGROUP(
  C1 AS "column1", C2 AS "column2"
ORDER BY C1 OPTION ROW "entry" ROOT "document")
FROM T1
```

```xml
<document>
  <entry>
    <column1>1</column1>
    <column2>2</column2>
  </entry>
  <entry>
    <column1>1</column1>
  </entry>
  <entry>
    <column2>2</column2>
  </entry>
</document>
```

Scalar functions

A scalar function can be used wherever an expression can be used. However, the restrictions that apply to the use of expressions and aggregate functions also apply when an expression or aggregate function is used within a scalar function. For example, the argument of a scalar function can be an aggregate function only if an aggregate function is allowed in the context in which the scalar function is used.

The restrictions on the use of aggregate functions do not apply to scalar functions, because a scalar function is applied to a single value rather than to a set of values.

The result of the following SELECT statement has as many rows as there are employees in department D01:

```sql
SELECT EMPNO, LASTNAME, YEAR(CURRENT DATE - BRTHDATE)
FROM EMPLOYEE
WHERE WORKDEPT = 'D01'
```
Scalar functions

Scalar functions can be qualified with a schema name (for example, SYSIBM.CHAR(123)).

In a Unicode database, all scalar functions that accept a character or graphic string will accept any string types for which conversion is supported.
ABS or ABSVAL

ABS or ABSVAL

\[
\text{ABS} \quad \text{(expression)} \\
\text{ABSVAL}
\]

The schema is SYSIBM.

The SYSFUN version of the ABS (or ABSVAL) function continues to be available.

Returns the absolute value of the argument. The argument can be any built-in numeric data type.

The result has the same data type and length attribute as the argument. The result can be null; if the argument is null, the result is the null value. If the argument is the maximum negative value for SMALLINT, INTEGER or BIGINT, the result is an overflow error.

Notes

Results involving DECFLOAT special values: For decimal floating-point values, the special values are treated as follows:

- \( \text{ABS}(\text{NaN}) \) and \( \text{ABS}(-\text{NaN}) \) return NaN.
- \( \text{ABS}(\text{Infinity}) \) and \( \text{ABS}(-\text{Infinity}) \) return Infinity.
- \( \text{ABS}(\text{sNaN}) \) and \( \text{ABS}(-\text{sNaN}) \) return sNaN.

Example

\( \text{ABS}(-51234) \)

returns an INTEGER with a value of 51234.
The schema is SYSIBM. (The SYSFUN version of the ACOS function continues to be available.)

Returns the arccosine of the argument as an angle expressed in radians.

The argument can be of any built-in numeric data type (except for DECFLOAT). It is converted to a double-precision floating-point number for processing by the function.

The result of the function is a double-precision floating-point number. The result can be null if the argument can be null or the database is configured with `dft_sqlmathwarn` set to YES; the result is the null value if the argument is null.

Example:

Assume that the host variable ACOSINE is a DECIMAL(10,9) host variable with a value of 0.070737202.

```sql
SELECT ACOS(:ACOSINE)
FROM SYSIBM.SYSDUMMY1
```

This statement returns the approximate value 1.49.
ADD_MONTHS

The schema is SYSIBM.

The ADD_MONTHS function returns a datetime value that represents \textit{expression} plus a specified number of months.

\textit{expression}  
An expression that specifies the starting date. The expression must return a value of one of the following built-in data types: a DATE or a TIMESTAMP.

\textit{numeric-expression}  
An expression that returns a value of any built-in numeric data type. If the value is not of type INTEGER, it is implicitly cast to INTEGER before evaluating the function. The \textit{numeric-expression} specifies the number of months to add to the starting date specified by \textit{expression}. A negative numeric value is allowed.

The result of the function has the same data type as \textit{expression}, unless \textit{expression} is a string, in which case the result data type is DATE. The result can be null; if any argument is null, the result is the null value.

If \textit{expression} is the last day of the month or if the resulting month has fewer days than the day component of \textit{expression}, the result is the last day of the resulting month. Otherwise, the result has the same day component as \textit{expression}. Any hours, minutes, seconds or fractional seconds information included in \textit{expression} is not changed by the function.

\textbf{Examples}

- Assume today is January 31, 2007. Set the host variable ADD_MONTH with the last day of January plus 1 month.
  
  \texttt{SET:ADD_MONTH = ADD_MONTHS(LAST\_DAY(CURRENT\_DATE), 1);}
  
  The host variable ADD_MONTH is set with the value representing the end of February, 2007-02-28.

- Assume DATE is a host variable with the value July 27, 1965. Set the host variable ADD_MONTH with the value of that day plus 3 months.
  
  \texttt{SET:ADD_MONTH = ADD\_MONTH(:DATE,3);} 
  
  The host variable ADD_MONTH is set with the value representing the day plus 3 months, 1965-10-27.

- It is possible to achieve similar results with the ADD_MONTHS function and date arithmetic. The following examples demonstrate the similarities and contrasts.
  
  \texttt{SET:DATEHV = DATE('2008-2-28') + 4 MONTHS;}  
  \texttt{SET:DATEHV = ADD\_MONTHS('2008-2-28', 4);}  
  
  In both cases, the host variable DATEHV is set with the value ‘2008-06-28’.

  Now consider the same examples but with the date ‘2008-2-29’ as the argument.

  \texttt{SET:DATEHV = DATE('2008-2-29') + 4 MONTHS;}  
  \texttt{SET:DATEHV = ADD\_MONTHS('2008-2-29', 4);}  
  
  The host variable DATEHV is set with the value ‘2008-06-29’.

  \texttt{SET:DATEHV = ADD\_MONTHS('2008-2-29', 4);}  
  
  The host variable DATEHV is set with the value ‘2008-06-30’.
In this case, the ADD_MONTHS function returns the last day of the month, which is June 30, 2008, instead of June 29, 2008. The reason is that February 29 is the last day of the month. So, the ADD_MONTHS function returns the last day of June.
The schema is SYSIBM.

The ARRAY_DELETE function deletes elements from an array.

array-variable
An SQL variable, SQL parameter, or global variable of an array type, or a CAST specification of a parameter marker to an array type.

array-index1
An expression that results in a value that can be assigned to the data type of the array index. If array-variable is an ordinary array, array-index1 must be the null value.

array-index2
An expression that results in a value that can be assigned to the data type of the array index. If array-variable is an ordinary array, array-index2 must be the null value. If array-index2 is specified and is a non-null value, then array-index1 must be a non-null value that is less than the value of array-index2 (SQLSTATE 42815).

The result of the function has the same data type as array-variable. If the optional arguments are not specified or they are the null value, all of the elements of array-variable are deleted and the cardinality of the result array value is 0. If only array-index1 is specified with a non-null value, the array element at index value array-index1 is deleted. If array-index2 is also specified with a non-null value, then the elements ranging from index value array-index1 to array-index2 (inclusive) are deleted.

The result can be null; if array-variable is null, the result is the null value.

Notes

- The ARRAY_DELETE function can be used only on the right side of an assignment statement in contexts where arrays are supported.

Examples

- Delete all the elements from the ordinary array variable RECENT_CALLS of array type PHONENUMBERS.
  SET RECENT_CALLS = ARRAY_DELETE(RECENT_CALLS)

- A supplier has discontinued some of their products. Delete the elements from the associative array variable FLOOR_TILES of array type PRODUCTS from index value 'PK5100' to index value 'PS2500'.
  SET FLOOR_TILES = ARRAY_DELETE(FLOOR_TILES, 'PK5100', 'PS2500')
The schema is SYSIBM.

The \texttt{ARRAY\_FIRST} function returns the minimum array index value of the array. 

\texttt{array-variable}

An SQL variable, SQL parameter, or global variable of an array type, or a 
CAST specification of a parameter marker to an array type.

The data type of the result is the data type of the array index, which is INTEGER 
for an ordinary array. If \texttt{array-variable} is not null and the cardinality of the array is 
greater than zero, the value of the result is the minimum array index value, which 
is 1 for an ordinary array.

The result can be null; if \texttt{array-variable} is null or the cardinality of the array is zero, 
the result is the null value.

\textbf{Examples}

\begin{itemize}
\item Return the first index value in the ordinary array variable \texttt{SPECIALNUMBERS} to the 
SQL variable \texttt{E\_CONSTIDX}.
\texttt{SET E\_CONSTIDX = ARRAY\_FIRST(SPECIALNUMBERS)}

The result is 1.
\item Given the associative array variable \texttt{PHONELIST} with index values and phone 
numbers: ‘Home’ is ‘4163053745’, ‘Work’ is ‘4163053746’, and ‘Mom’ is 
‘416-4789683’, assign the value of the minimum index in the array to the 
character string variable named \texttt{X}.
\texttt{SET X = ARRAY\_FIRST(PHONELIST)}

The value of ‘Home’ is assigned to \texttt{X}. Access the element value associated with 
index value ‘Home’ and assign it to the SQL variable \texttt{NUMBER\_TO\_CALL}:
\texttt{SET NUMBER\_TO\_CALL = PHONELIST[X]}
\end{itemize}
The schema is SYSIBM.

The `ARRAY_LAST` function returns the maximum array index value of the array.

`array-variable`

An SQL variable, SQL parameter, or global variable of an array type, or a CAST specification of a parameter marker to an array type.

The data type of the result is the data type of the array index, which is INTEGER for an ordinary array. If `array-variable` is not null and the cardinality of the array is greater than zero, the value of the result is the maximum array index value, which is the cardinality of the array for an ordinary array.

The result can be null; if `array-variable` is null or the cardinality of the array is zero, the result is the null value.

**Examples**

- Return the last index value in the ordinary array variable `SPECIALNUMBERS` to the SQL variable `PI_CONSTIDX`.

  ```sql
  SET PI_CONSTIDX = ARRAY_LAST(SPECIALNUMBERS)
  ```

  The result is 10.

- Given the associative array variable `PHONELIST` with index values and phone numbers: 'Home' is '4163053745', 'Work' is '4163053746', and 'Mom' is '4164789683', assign the value of the maximum index in the array to the character string variable named `X`.

  ```sql
  SET X = ARRAY_LAST(PHONELIST)
  ```

  The value of 'Work' is assigned to `X`. Access the element value associated with index value 'Work' and assign it to the SQL variable `NUMBER_TO_CALL`:

  ```sql
  SET NUMBER_TO_CALL = PHONELIST[X]
  ```
The schema is SYSIBM.

The ARRAY_NEXT function returns the next larger array index value for an array relative to the specified array index argument.

array-variable
An SQL variable, SQL parameter, or global variable of an array type, or a CAST specification of a parameter marker to an array type.

array-index
Specifies a value that is assignable to the data type of the index of the array. Valid values include any valid value for the data type.

The result is the next larger array index value defined in the array relative to the specified array-index value. If array-index is less than the minimum index array value in the array, the result is the first array index value defined in the array.

The data type of the result of the function is the data type of the array index. The result can be null; if either argument is null, the cardinality of the first argument is zero, or the value of array-index is greater than or equal to the value of the last index in the array, the result is the null value.

Examples

- Return the next index value after the 9th index position in the ordinary array variable SPECIALNUMBERS to the SQL variable NEXT_CONSTIDX.

  SET NEXT_CONSTIDX = ARRAY_NEXT(SPECIALNUMBERS, 9)

  The result is 10.

- Given the associative array variable PHONELIST with index values and phone numbers: 'Home' is '4163053745', 'Work' is '4163053746', and 'Mom' is '416-4789683', assign the value of the index in the array that is the next index after index value 'Dad', which does not exist for the array value, to the character string variable named X:

  SET X = ARRAY_NEXT(PHONELIST, 'Dad')

  The value of 'Home' is assigned to X, since the value 'Dad' is a value smaller than any index value for the array variable. Assign the value of the index in the array that is the next index after index value 'Work':

  SET X = ARRAY_NEXT(PHONELIST, 'Work')

  The null value is assigned to X.
The schema is SYSIBM.

The ARRAY_PRIOR function returns the next smaller array index value for an array relative to the specified array index argument.

**array-variable**

An SQL variable, SQL parameter, or global variable of an array type, or a CAST specification of a parameter marker to an array type.

**array-index**

Specifies a value that is assignable to the data type of the index of the array.

Valid values include any valid value for the data type.

The result is the next smaller array index value defined in the array relative to the specified **array-index** value. If **array-index** is greater than the maximum index array value in the array, the result is the last array index value defined in the array.

The data type of the result of the function is the data type the array index. The result can be null; if either argument is null, the cardinality of the first argument is zero, or the value of **array-index** is less than or equal to the value of the first index in the array, the result is the null value.

**Examples**

- Return the previous index value before the 2nd index position in the ordinary array variable SPECIALNUMBERS to the SQL variable PREV_CONSTIDX.

  ```sql
  SET PREV_CONSTIDX = ARRAY_PRIOR(SPECIALNUMBERS, 2)
  ```

  The result is 1.

- Given the associative array variable PHONELIST with index values and phone numbers: 'Home' is '4163053745', 'Work' is '4163053746', and 'Mom' is '416-4789683', assign the value of the index in the array that is the previous index before index value 'Work' to the character string variable named X:

  ```sql
  SET X = ARRAY_PRIOR(PHONELIST, 'Work')
  ```

  The value of 'Mom' is assigned to X. Assign the value of the index in the array that is the previous index before index value 'Home':

  ```sql
  SET X = ARRAY_PRIOR(PHONELIST, 'Home')
  ```

  Thenull value is assigned to X.
The schema is SYSFUN.

Returns the ASCII code value of the leftmost character of the argument as an integer.

The argument can be of any built-in character string type. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed. For a VARCHAR, the maximum length is 4,000 bytes, and for a CLOB, the maximum length is 1,048,576 bytes. LONG VARCHAR is converted to CLOB for processing by the function.

The result of the function is always INTEGER.

The result can be null; if the argument is null, the result is the null value.
The ASIN function returns the arcsine of a value in radians.

**Syntax:**
```
ASIN(expression)
```

- **Schema:** SYSIBM
- **Usage:** Available in all database configurations.

**Description:**
The ASIN function accepts any numeric type except DECFLOAT and returns the arcsine of the given value as a double-precision floating-point number. If the input is null, the result is also null. Setting `dft_sqlmathwarn` to YES or enabling database configurations with dft_sqlmathwarn will trigger a warning. Setting `dft_sqlmathwarn` to YES ensures that the function is not available if the argument is null or the database is configured with dft_sqlmathwarn.
ATAN

ATAN

\( \text{ATAN}(\text{expression}) \)

The schema is SYSIBM. (The SYSFUN version of the ATAN function continues to be available.)

Returns the arctangent of the argument as an angle expressed in radians.

The argument can be of any built-in numeric data type (except for DECFLOAT). It is converted to a double-precision floating-point number for processing by the function.

The result of the function is a double-precision floating-point number. The result can be null if the argument can be null or the database is configured with `dft_sqlmathwarn` set to YES; the result is the null value if the argument is null.
ATAN2

\[
\text{ATAN2}(\text{expression}, \text{expression})
\]

The schema is SYSIBM. (The SYSFUN version of the ATAN2 function continues to be available.)

Returns the arctangent of x and y coordinates as an angle expressed in radians. The x and y coordinates are specified by the first and second arguments, respectively.

The first and the second arguments can be of any built-in numeric data type (except for DECFLOAT). Both are converted to a double-precision floating-point number for processing by the function.

The result of the function is a double-precision floating-point number. The result can be null if the argument can be null or the database is configured with \texttt{dft_sqlmathwarn} set to YES; the result is the null value if the argument is null.
The schema is SYSIBM.

Returns the hyperbolic arctangent of the argument, where the argument is an angle expressed in radians.

The argument can be of any built-in numeric data type (except for DECFLOAT). It is converted to a double-precision floating-point number for processing by the function.

The result of the function is a double-precision floating-point number. The result can be null if the argument can be null or the database is configured with dft_sqlmathwarn set to YES; the result is the null value if the argument is null.
BIGINT

Numeric to Big Integer

\[ \text{BIGINT} \left( \text{numeric-expression} \right) \]

String to Big Integer

\[ \text{BIGINT} \left( \text{string-expression} \right) \]

Datetime to Big Integer

\[ \text{BIGINT} \left( \text{datetime-expression} \right) \]

The schema is SYSIBM.

The BIGINT function returns a 64-bit integer representation of:
- A number
- A string representation of a number
- A datetime value

**Numeric to Big Integer**

\textit{numeric-expression}

An expression that returns a value of any built-in numeric data type.

The result is the same number that would occur if the argument were assigned to a big integer column or variable. The fractional part of the argument is truncated. If the whole part of the argument is not within the range of big integers, an error is returned (SQLSTATE 22003).

**String to Big Integer**

\textit{string-expression}

An expression that returns a value that is a character-string or Unicode graphic-string representation of a number with a length not greater than the maximum length of a character constant.

The result is the same number that would result from \textit{CAST(string-expression AS BIGINT)}. Leading and trailing blanks are eliminated and the resulting string must conform to the rules for forming an integer, decimal, floating-point, or decimal floating-point constant (SQLSTATE 22018). If the whole part of the argument is not within the range of big integers, an error is returned (SQLSTATE 22003). The data type of \textit{string-expression} must not be CLOB or DBCLOB (SQLSTATE 42884).

**Datetime to Big Integer**

\textit{datetime-expression}

An expression that is of one of the following data types:
- DATE. The result is a BIGINT value representing the date as \textit{yyyymmdd}.
- TIME. The result is a BIGINT value representing the time as \textit{hmmmss}.
• TIMESTAMP: The result is a BIGINT value representing the timestamp as `yyyyymmddhhmmss`. The fractional seconds portion of the timestamp value is not included in the result.

The result of the function is a big integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

**Note:** The CAST specification should be used to increase the portability of applications. For more information, see “CAST specification”.

**Examples**

• From ORDERS_HISTORY table, count the number of orders and return the result as a big integer value.

```
SELECT BIGINT (COUNT_BIG(*))
FROM ORDERS_HISTORY
```

• Using the EMPLOYEE table, select the EMPNO column in big integer form for further processing in the application.

```
SELECT BIGINT (EMPNO) FROM EMPLOYEE
```

• Assume that the column RECEIVED (whose data type is TIMESTAMP) has an internal value equivalent to ‘1988-12-22-14.07.21.136421’.

```
BIGINT(RECEIVED)
```

results in the value 19881222140721.

• Assume that the column STARTTIME (whose data type is TIME) has an internal value equivalent to ‘12:03:04’.

```
BIGINT(STARTTIME)
```

results in the value 120304.
The schema is SYSIBM.

These bitwise functions operate on the “two’s complement” representation of the integer value of the input arguments and return the result as a corresponding base 10 integer value in a data type based on the data type of the input arguments.

Table 43. The bit manipulation functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>A bit in the two’s complement representation of the result is:</th>
</tr>
</thead>
<tbody>
<tr>
<td>BITAND</td>
<td>Performs a bitwise AND operation.</td>
<td>1 only if the corresponding bits in both arguments are 1.</td>
</tr>
<tr>
<td>BITANDNOT</td>
<td>Clears any bit in the first argument that is in the second argument.</td>
<td>Zero if the corresponding bit in the second argument is 1; otherwise, the result is copied from the corresponding bit in the first argument.</td>
</tr>
<tr>
<td>BITOR</td>
<td>Performs a bitwise OR operation.</td>
<td>1 unless the corresponding bits in both arguments are zero.</td>
</tr>
<tr>
<td>BITXOR</td>
<td>Performs a bitwise exclusive OR operation.</td>
<td>1 unless the corresponding bits in both arguments are the same.</td>
</tr>
<tr>
<td>BITNOT</td>
<td>Performs a bitwise NOT operation.</td>
<td>Opposite of the corresponding bit in the argument.</td>
</tr>
</tbody>
</table>

The arguments must be integer values represented by the data types SMALLINT, INTEGER, BIGINT, or DECFLOAT. Arguments of type DECIMAL, REAL, or DOUBLE are cast to DECFLOAT. The value is truncated to a whole number.

The bit manipulation functions can operate on up to 16 bits for SMALLINT, 32 bits for INTEGER, 64 bits for BIGINT, and 113 bits for DECFLOAT. The range of supported DECFLOAT values includes integers from \(-2^{112}\) to \(2^{112} -1\), and special values such as NaN or INFINITY are not supported (SQLSTATE 42815). If the two arguments have different data types, the argument supporting fewer bits is cast to a value with the data type of the argument supporting more bits. This cast impacts the bits that are set for negative values. For example, -1 as a SMALLINT value has 16 bits set to 1, which when cast to an INTEGER value has 32 bits set to 1.

The result of the functions with two arguments has the data type of the argument that is highest in the data type precedence list for promotion. If either argument is
**BITAND, BITANDNOT, BITOR, BITXOR, and BITNOT**

DECFLOAT, the data type of the result is DECFLOAT(34). If either argument can be null, the result can be null; if either argument is null, the result is the null value.

The result of the BITNOT function has the same data type as the input argument, except that DECIMAL, REAL, DOUBLE, or DECFLOAT(16) returns DECFLOAT(34). If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Due to differences in internal representation between data types and on different hardware platforms, using functions (such as HEX) or host language constructs to view or compare internal representations of BIT function results and arguments is data type-dependent and not portable. The data type- and platform-independent way to view or compare BIT function results and arguments is to use the actual integer values.

Use of the BITXOR function is recommended to toggle bits in a value. Use the BITANDNOT function to clear bits. BITANDNOT(val, pattern) operates more efficiently than BITAND(val, BITNOT(pattern)).

Examples:

The following examples are based on an ITEM table with a PROPERTIES column of type INTEGER.

- Return all items for which the third property bit is set.
  
  ```sql
  SELECT ITEMID FROM ITEM
  WHERE BITAND(PROPERTIES, 4) = 4
  ```

- Return all items for which the fourth or the sixth property bit is set.
  
  ```sql
  SELECT ITEMID FROM ITEM
  WHERE BITAND(PROPERTIES, 40) <> 0
  ```

- Clear the twelfth property of the item whose ID is 3412.
  
  ```sql
  UPDATE ITEM
  SET PROPERTIES = BITANDNOT(PROPERTIES, 2048)
  WHERE ITEMID = 3412
  ```

- Set the fifth property of the item whose ID is 3412.
  
  ```sql
  UPDATE ITEM
  SET PROPERTIES = BITOR(PROPERTIES, 16)
  WHERE ITEMID = 3412
  ```

- Toggle the eleventh property of the item whose ID is 3412.
  
  ```sql
  UPDATE ITEM
  SET PROPERTIES = BITXOR(PROPERTIES, 1024)
  WHERE ITEMID = 3412
  ```

- Switch all the bits in a 16-bit value that has only the second bit on.
  
  ```sql
  VALUES BITNOT(CAST(2 AS SMALLINT))
  ```

returns -3 (with a data type of SMALLINT).
The schema is SYSIBM.

The BLOB function returns a BLOB representation of a string of any type.

**string-expression**

A string-expression whose value can be a character string, graphic string, or a binary string.

**integer**

An integer value specifying the length attribute of the resulting BLOB data type. If integer is not specified, the length attribute of the result is the same as the length of the input, except where the input is graphic. In this case, the length attribute of the result is twice the length of the input.

The result of the function is a BLOB. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Examples

- Given a table with a BLOB column named TOPOGRAPHIC_MAP and a VARCHAR column named MAP_NAME, locate any maps that contain the string 'Pellow Island' and return a single binary string with the map name concatenated in front of the actual map.

  ```sql
  SELECT BLOB(MAP_NAME CONCAT ': ') CONCAT TOPOGRAPHIC_MAP
  FROM ONTARIO_SERIES_4
  WHERE TOPOGRAPHIC_MAP LIKE BLOB('%Pellow Island%')
  ```
CARDINALITY

The schema is SYSIBM.

The CARDINALITY function returns a value of type BIGINT representing the number of elements of an array.

array-variable
An SQL variable, SQL parameter, or global variable of an array type, or a CAST specification of a parameter marker to an array type.

For an ordinary array, the value returned by the CARDINALITY function is the highest array index for which the array has an assigned element. This includes elements that have been assigned the null value. For an associative array, the value returned by the CARDINALITY function is the actual number of unique array index values defined in array-variable.

The function returns 0 if the array is empty. The result can be null; if the argument is null, the result is the null value.

Examples
• Return the number of calls that have been stored in the recent calls list so far:
  SET HOWMANYCALLS = CARDINALITY(RECENT_CALLS)

  The SQL variable HOWMANYCALLS contains the value 3.

• Assume that the associative array variable CAPITALS of array type CAPITALSARRAY contains all of the capitals for the 10 provinces and 3 territories in Canada as well as the capital of the country, Ottawa. Return the cardinality of the array variable:
  SET NUMCAPITALS = CARDINALITY(CAPITALS)

  The SQL variable NUMCAPITALS contains the value 14.
CEILING or CEIL

The schema is SYSIBM. (The SYSFUN version of the CEILING function continues to be available.)

Returns the smallest integer value greater than or equal to the argument.

The argument can be of any built-in numeric type. The result of the function has the same data type and length attribute as the argument except that the scale is 0 if the argument is DECIMAL. For example, an argument with a data type of DECIMAL(5,5) returns DECIMAL(5,0).

The result can be null if the argument can be null or if the argument is not a decimal floating-point number and the database is configured with dft_sqlmathwarn set to YES; the result is the null value if the argument is null.

Notes

Results involving DECFLOAT special values: For decimal floating-point values, the special values are treated as follows:
• CEILING(NaN) returns NaN.
• CEILING(-NaN) returns -NaN.
• CEILING(Infinity) returns Infinity.
• CEILING(-Infinity) returns -Infinity.
• CEILING(sNaN) returns NaN and a warning.
• CEILING(-sNaN) returns -NaN and a warning.

Syntax alternatives: CEIL can be specified in place of CEILING.
CHAR

Integer to character

```
CHAR(integer-expression)
```

Decimal to character

```
CHAR(decimal-expression,decimal-character)
```

Floating-point to character

```
CHAR(floating-point-expression,decimal-character)
```

Decimal floating-point to character

```
CHAR(decimal-floating-point-expression,decimal-character)
```

Character to character

```
CHAR(character-expression,integer)
```

Graphic to character

```
CHAR(graphic-expression,integer)
```

Datetime to character

```
CHAR(datetime-expression,ISO,USA,eur,JIS,local)
```

The schema is SYSIBM. The function name cannot be specified as a qualified name when keywords are used in the function signature. The SYSFUN.CHAR(floating-point-expression) signature continues to be available. In this case, the decimal character is locale sensitive, and therefore returns either a period or a comma, depending on the locale of the database server.

The CHAR function returns a fixed-length character string representation of:

- An integer number, if the first argument is a SMALLINT, INTEGER, or BIGINT
- A decimal number, if the first argument is a decimal number
The result of the function is a fixed-length character string. If the first argument can be null, the result can be null. If the first argument is null, the result is the null value.

**Integer to character**

\[ \text{integer-expression} \]

An expression that returns a value that is of an integer data type (SMALLINT, INTEGER, or BIGINT).

The result is a fixed-length character string representation of \( \text{integer-expression} \) in the form of an SQL integer constant. The result consists of \( n \) characters, which represent the significant digits in the argument, and is preceded by a minus sign if the argument is negative. The result is left justified.

- If the first argument is a small integer, the length of the result is 6.
- If the first argument is a large integer, the length of the result is 11.
- If the first argument is a big integer, the length of the result is 20.

If the number of bytes in the result is less than the defined length of the result, the result is padded on the right with single-byte blanks.

The code page of the result is the code page of the section.

**Decimal to character**

\[ \text{decimal-expression} \]

An expression that returns a value that is a decimal data type. If a different precision and scale are required, the DECIMAL scalar function can be used first to make the change.

\[ \text{decimal-character} \]

Specifies the single-byte character constant that is used to delimit the decimal digits in the result character string. The character constant cannot be a digit, the plus sign (+), the minus sign (-), or a blank (SQLSTATE 42815). The default is the period (.) character.

The result is a fixed-length character string representation of \( \text{decimal-expression} \) in the form of an SQL decimal constant. The length of the result is \( 2+p \), where \( p \) is the precision of \( \text{decimal-expression} \). Leading zeros
are not included. Trailing zeros are included. If `decimal-expression` is negative, the first character of the result is a minus sign; otherwise, the first character is a digit or the decimal character. If the scale of `decimal-expression` is zero, the decimal character is not returned. If the number of bytes in the result is less than the defined length of the result, the result is padded on the right with single-byte blanks.

The code page of the result is the code page of the section.

**Floating-point to character**

`floating-point-expression`

An expression that returns a value that is a floating-point data type (DOUBLE or REAL).

`decimal-character`

Specifies the single-byte character constant that is used to delimit the decimal digits in the result character string. The character constant cannot be a digit, the plus sign (+), the minus sign (-), or a blank (SQLSTATE 42815). The default is the period (.) character.

The result is a fixed-length character string representation of `floating-point-expression` in the form of an SQL floating-point constant. The length of the result is 24. The result is the smallest number of characters that can represent the value of `floating-point-expression` such that the mantissa consists of a single digit other than zero followed by a period and a sequence of digits. If `floating-point-expression` is negative, the first character of the result is a minus sign; otherwise, the first character is a digit. If `floating-point-expression` is zero, the result is 0E0. If the number of bytes in the result is less than 24, the result is padded on the right with single-byte blanks.

The code page of the result is the code page of the section.

**Decimal floating-point to character**

`decimal-floating-point-expression`

An expression that returns a value that is a decimal floating-point data type (DECFLOAT).

`decimal-character`

Specifies the single-byte character constant that is used to delimit the decimal digits in the result character string. The character constant cannot be a digit, the plus sign (+), the minus sign (-), or a blank (SQLSTATE 42815). The default is the period (.) character.

The result is a fixed-length character string representation of `decimal-floating-point-expression` in the form of an SQL decimal floating-point constant. The length attribute of the result is 42. The result is the smallest number of characters that can represent the value of `decimal-floating-point-expression`. If `decimal-floating-point-expression` is negative, the first character of the result is a minus sign; otherwise, the first character is a digit. If `decimal-floating-point-expression` is zero, the result is 0. If the value of `decimal-floating-point-expression` is the special value Infinity, sNaN, or NaN, the strings 'INFINITY', 'SNAN', and 'NAN', respectively, are returned. If the special value is negative, the first character of the result is a minus sign. The decimal floating-point special value sNaN does not
result in warning when converted to a string. If the number of characters in the result is less than 42, the result is padded on the right with single-byte blanks.

The code page of the result is the code page of the section.

**Character to character**

*character-expression*

An expression that returns a value that is a built-in character string data type (CHAR, VARCHAR, or CLOB).

*integer*

The length attribute for the resulting fixed-length character string. The value must be between 0 and 254.

If the second argument is not specified:

- If the *character-expression* is the empty string constant, the length attribute of the result is 0.
- Otherwise, the length attribute of the result is the same as the length attribute of the first argument. If the actual length of the first argument (excluding trailing blanks) is greater than 254, an error is returned (SQLSTATE 22001).

The actual length of the result is the same as the length attribute of the result. If the length of the *character-expression* is less than the length of the result, the result is padded with blanks up to the length of the result. If the length of the *character-expression* is greater than the length attribute of the result, truncation is performed. A warning is returned (SQLSTATE 01004) unless the truncated characters were all blanks and the *character-expression* was not a CLOB.

If the length of the character expression is less than the length attribute of the result, the result is padded with blanks up to the length of the result. If the length of the character expression is greater than the length attribute of the result, the result is truncated. A warning is returned (SQLSTATE 01004), unless the truncated characters were all blanks, and the character expression was not a CLOB.

**Graphic to character**

*graphic-expression*

An expression that returns a value that is a built-in graphic string data type. (GRAPHIC, VARGRAPHIC, or DBCLOB).

*integer*

The length attribute for the resulting fixed-length character string. The value must be between 0 and 254.

If the second argument is not specified:

- If the *graphic-expression* is the empty string constant, the length attribute of the result is 0.
- Otherwise, the length attribute of the result is the same as MIN (254, 3 * length attribute of the first argument). If the actual length of the first argument (including trailing blanks) is greater than 254, an error is returned (SQLSTATE 22001).

The actual length of the result is the same as the length attribute of the result. If the length of the *graphic-expression* is less than the length of the result, the result is padded with blanks up to the length of the result.
result. If the length of the `graphic-expression` is greater than the length attribute of the result, truncation is performed with no warning returned.

**Datetime to character**

`datetime-expression`

An expression that is of one of the following data types:

**DATE** The result is the character string representation of the date in the format specified by the second argument. The length of the result is 10. An error is returned if the second argument is specified and is not a valid value (SQLSTATE 42703).

**TIME** The result is the character string representation of the time in the format specified by the second argument. The length of the result is 8. An error is returned if the second argument is specified and is not a valid value (SQLSTATE 42703).

**TIMESTAMP**

The result is the character string representation of the timestamp. If the data type of `datetime-expression` is `TIMESTAMP(0)`, the length of the result is 19. If the data type of `datetime-expression` is `TIMESTAMP(n)`, where `n` is between 1 and 12, the length of the result is 20+n. Otherwise, the length of the result is 26. The second argument is not applicable and must not be specified (SQLSTATE 42815).

The code page of the result is the code page of the section.

**Notes:**

- The CAST specification should be used to increase the portability of applications when the first argument is numeric, or the first argument is a string and the length argument is specified. For more information, see “CAST specification”.
- A binary string is allowed as the first argument to the function, and the resulting fixed-length string is a FOR BIT DATA character string, padded with blanks if necessary.
- **Decimal to character and leading zeros:** In versions previous to version 9.7, the result for decimal input to this function includes leading zeros and a trailing decimal character. The database configuration parameter `dec_to_char_fmt` can be set to “V95” to have this function return the version 9.5 result for decimal input. The default value of `dec_to_char_fmt` for new databases is “NEW”, which has this function return results which match the SQL standard casting rules and is consistent with results from the VARCHAR function.

**Examples**

- Assume that the PRSTDATE column has an internal value equivalent to 1988-12-25. The following function returns the value ‘12/25/1988’.

  `CHAR(PRSTDATE, USA)`

- Assume that the STARTING column has an internal value equivalent to 17:12:30, and that the host variable HOUR_DUR (decimal(6,0)) is a time duration with a value of 050000 (that is, 5 hours). The following function returns the value ‘5:12 PM’.

  `CHAR(STARTING, USA)`

The following function returns the value ‘10:12 PM’.

`CHAR(STARTING + :HOUR_DUR, USA)`
• Assume that the RECEIVED column (TIMESTAMP) has an internal value equivalent to the combination of the PRSTDATE and STARTING columns. The following function returns the value ‘1988-12-25-17.12.30.000000’.

```
CHAR(RECEIVED)
```

• The LASTNAME column is defined as VARCHAR(15). The following function returns the values in this column as fixed-length character strings that are 10 bytes long. LASTNAME values that are more than 10 bytes long (excluding trailing blanks) are truncated and a warning is returned.

```
SELECT CHAR(LASTNAME, 10) FROM EMPLOYEE
```

• The EDLEVEL column is defined as SMALLINT. The following function returns the values in this column as fixed-length character strings. An EDLEVEL value of 18 is returned as the CHAR(6) value ‘18’ followed by four blanks.

```
SELECT CHAR(EDLEVEL) FROM EMPLOYEE
```

• The SALARY column is defined as DECIMAL with a precision of 9 and a scale of 2. The current value (18357.50) is to be displayed with a comma as the decimal character (18357,50). The following function returns the value ‘18357,50’ followed by three blanks.

```
CHAR(SALARY, ',', )
```

• Values in the SALARY column are to be subtracted from 20000.25 and displayed with the default decimal character. The following function returns the value ‘-0001642.75’ followed by three blanks.

```
CHAR(20000.25 - SALARY)
```

• Assume that the host variable SEASONS_TICKETS is defined as INTEGER and has a value of 10000. The following function returns the value ‘10000.00’.

```
CHAR(DECIMAL(:SEASONS_TICKETS,7,2))
```

• Assume that the host variable DOUBLE_NUM is defined as DOUBLE and has a value of -987.654321E-35. The following function returns the value ‘-9.87654321E-33’ followed by nine trailing blanks because the result data type is CHAR(24).

```
CHAR(:DOUBLE_NUM)
```
CHARACTER_LENGTH

The schema is SYSIBM.

The CHARACTER_LENGTH function returns the length of expression in the specified string unit.

expression
   An expression that returns a value of a built-in character or graphic string.

CODEUNITS16, CODEUNITS32, or OCTETS
   Specifies the string unit of the result. CODEUNITS16 specifies that the result is to be expressed in 16-bit UTF-16 code units. CODEUNITS32 specifies that the result is to be expressed in 32-bit UTF-32 code units. OCTETS specifies that the result is to be expressed in bytes.

   If a string unit is specified as CODEUNITS16 or CODEUNITS32, and expression is a binary string or bit data, an error is returned (SQLSTATE 428GC). If a string unit is specified as OCTETS and expression is a binary string, an error is returned (SQLSTATE 42815). For more information about CODEUNITS16, CODEUNITS32, and OCTETS, see “String units in built-in functions” in “Character strings”.

The result of the function is a large integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The length of character and graphic strings includes trailing blanks. The length of varying-length strings is the actual length and not the maximum length.

Examples:
   • Assume that NAME is a VARCHAR(128) column, encoded in Unicode UTF-8, that contains the value ‘Jürgen’. The following two queries return the value 6:

```
SELECT CHARACTER_LENGTH(NAME, CODEUNITS32)
FROM T1 WHERE NAME = 'Jürgen'
```

```
SELECT CHARACTER_LENGTH(NAME, CODEUNITS16)
FROM T1 WHERE NAME = 'Jürgen'
```

The following two queries return the value 7:

```
SELECT CHARACTER_LENGTH(NAME, OCTETS)
FROM T1 WHERE NAME = 'Jürgen'
```

```
SELECT LENGTH(NAME)
FROM T1 WHERE NAME = 'Jürgen'
```

   • The following examples work with the Unicode string ‘&N~AB’, where ‘&’ is the musical symbol G clef character, and ‘~’ is the combining tilde character. This string is shown below in different Unicode encoding forms:

<table>
<thead>
<tr>
<th></th>
<th>&amp;</th>
<th>N</th>
<th>~</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>UTF-8</td>
<td>X'09D849E'</td>
<td>X'4E'</td>
<td>X'CC83'</td>
<td>X'41'</td>
<td>X'42'</td>
</tr>
<tr>
<td>UTF-16BE</td>
<td>X'D834DD1E'</td>
<td>X'004E'</td>
<td>X'0303'</td>
<td>X'0041'</td>
<td>X'0042'</td>
</tr>
<tr>
<td>UTF-32BE</td>
<td>X'0001D11E'</td>
<td>X'0000004E'</td>
<td>X'00000303'</td>
<td>X'00000041'</td>
<td>X'00000042'</td>
</tr>
</tbody>
</table>
Assume that the variable UTF8_VAR contains the UTF-8 representation of the string,

```sql
SELECT CHARACTER_LENGTH(UTF8_VAR, CODEUNITS16),
      CHARACTER_LENGTH(UTF8_VAR, CODEUNITS32),
      CHARACTER_LENGTH(UTF8_VAR, OCTETS)
FROM SYSIBM.SYSDUMMY1
```

returns the values 6, 5, and 9, respectively.

Assume that the variable UTF16_VAR contains the UTF-16BE representation of the string.

```sql
SELECT CHARACTER_LENGTH(UTF16_VAR, CODEUNITS16),
      CHARACTER_LENGTH(UTF16_VAR, CODEUNITS32),
      CHARACTER_LENGTH(UTF16_VAR, OCTETS)
FROM SYSIBM.SYSDUMMY1
```

returns the values 6, 5, and 12, respectively.
The schema is SYSFUN.

Returns the character that has the ASCII code value specified by the argument. If expression is 0, the result is the blank character (X'20').

The argument can be either INTEGER or SMALLINT. The value of the argument should be between 0 and 255; otherwise, the return value is the character that has the ASCII code value corresponding to 255.

The result of the function is CHAR(1). The result can be null; if the argument is null, the result is the null value.
The schema is SYSIBM.

The CLOB function returns a CLOB representation of a character string type. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string data type before the function is executed.

character-string-expression
An expression that returns a value that is a character string. The expression cannot be a character string defined as FOR BIT DATA (SQLSTATE 42846).

integer
An integer value specifying the length attribute of the resulting CLOB data type. The value must be between 0 and 2,147,483,647. If a value for integer is not specified, the length of the result is the same as the length of the first argument.

The result of the function is a CLOB. If the argument can be null, the result can be null; if the argument is null, the result is the null value.
The schema is SYSIBM.

COALESCE returns the first argument that is not null.

The arguments are evaluated in the order in which they are specified, and the result of the function is the first argument that is not null. The result can be null only if all the arguments can be null, and the result is null only if all the arguments are null. The selected argument is converted, if necessary, to the attributes of the result.

The arguments must be compatible. They can be of either a built-in or user-defined data type. (This function cannot be used as a source function when creating a user-defined function. Because this function accepts any compatible data types as arguments, it is not necessary to create additional signatures to support user-defined distinct types.)

Examples:

- When selecting all the values from all the rows in the DEPARTMENT table, if the department manager (MGRNO) is missing (that is, null), then return a value of 'ABSENT'.

  ```sql
  SELECT DEPTNO, DEPTNAME, COALESCE(MGRNO, 'ABSENT'), ADMRDEPT
  FROM DEPARTMENT
  ```

- When selecting the employee number (EMPNO) and salary (SALARY) from all the rows in the EMPLOYEE table, if the salary is missing (that is, null), then return a value of zero.

  ```sql
  SELECT EMPNO, COALESCE(SALARY, 0)
  FROM EMPLOYEE
  ```
The schema is SYSIBM.

The COLLATION_KEY_BIT function returns a VARCHAR FOR BIT DATA string representing the collation key of the string-expression in the specified collation-name.

The results of COLLATION_KEY_BIT for two strings can be binary compared to determine their order within the specified collation-name. For the comparison to be meaningful, the results used must be from the same collation-name.

**string-expression**

An expression that returns a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC string for which the collation key should be determined. If string-expression is a CHAR or VARCHAR, the expression must not be FOR BIT DATA (SQLSTATE 429BM).

If string-expression is not in UTF-16, this function performs code page conversion of string-expression to UTF-16. If the result of the code page conversion contains at least one substitution character, this function will return a collation key of the UTF-16 string with the substitution character(s) and the warning flag SQLWARN8 in the SQLCA will be set to 'W'.

**collation-name**

A character constant that specifies the collation to use when determining the collation key. The value of collation-name is not case sensitive and must be one of the “Unicode Collation Algorithm-based collations” in Globalization Guide or “language-aware collations for Unicode data” in Globalization Guide (SQLSTATE 42616).

**length**

An expression that specifies the length attribute of the result in bytes. If specified, length must be an integer between 1 and 32672 (SQLSTATE 42815).

If a value for length is not specified, the length of the result is determined as follows:

<table>
<thead>
<tr>
<th>String Argument Data Type</th>
<th>Result Data Type Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR(n) or VARCHAR(n)</td>
<td>Minimum of 12n bytes and 32672 bytes</td>
</tr>
<tr>
<td>GRAPHIC(n) or VARGRAPHIC(n)</td>
<td>Minimum of 12n bytes and 32672 bytes</td>
</tr>
</tbody>
</table>

Regardless of whether length is specified or not, if the length of the collation key is longer than the length of the result data type, an error is returned (SQLSTATE 42815). The actual result length of the collation key is approximately six times the length of string-expression after it has been converted to UTF-16.

If string-expression is an empty string, the result is a valid collation key that can have a nonzero length.

If any argument can be null, the result can be null; if any argument is null, the result is the null value.
COLLATION_KEY_BIT

Examples:

The following query orders employees by their surnames using the language-aware collation for German in code page 923:

```
SELECT FIRSTNAME, LASTNAME
FROM EMPLOYEE
ORDER BY COLLATION_KEY_BIT (LASTNAME, 'SYSTEM_923_DE')
```

The following query uses a culturally correct comparison to find the departments of employees in the province of Québec:

```
SELECT E.WORKDEPT
FROM EMPLOYEE AS E INNER JOIN SALES AS S
ON COLLATION_KEY_BIT(E.LASTNAME, 'UCA400R1_LFR') = COLLATION_KEY_BIT(S.SALES_PERSON, 'UCA400R1_LFR')
WHERE S.REGION = 'Quebec'
```
COMPARE_DECFLOAT

COMPARE_DECFLOAT

The schema is SYSIBM.

The COMPARE_DECFLOAT function returns a SMALLINT value that indicates whether the two arguments are equal or unordered, or whether one argument is greater than the other.

expression1
An expression that returns a value of any built-in numeric data type. If the argument is not DEC_FLOAT(34), it is logically converted to DEC_FLOAT(34) for processing.

expression2
An expression that returns a value of any built-in numeric data type. If the argument is not DEC_FLOAT(34), it is logically converted to DEC_FLOAT(34) for processing.

The value of expression1 is compared with the value of expression2, and the result is returned according to the following rules:

• If both arguments are finite, the comparison is algebraic and follows the procedure for decimal floating-point subtraction. If the difference is exactly zero with either sign, the arguments are equal. If a nonzero difference is positive, the first argument is greater than the second argument. If a nonzero difference is negative, the first argument is less than the second.
• Positive zero and negative zero compare as equal.
• Positive infinity compares equal to positive infinity.
• Positive infinity compares greater than any finite number.
• Negative infinity compares equal to negative infinity.
• Negative infinity compares less than any finite number.
• Numeric comparison is exact. The result is determined for finite operands as if range and precision were unlimited. No overflow or underflow condition can occur.
• If either argument is NaN or sNaN (positive or negative), the result is unordered.

The result value is as follows:

• 0 if the arguments are exactly equal
• 1 if expression1 is less than expression2
• 2 if expression1 is greater than expression2
• 3 if the arguments are unordered

The result of the function is a SMALLINT value. If either argument can be null, the result can be null; if either argument is null, the result is the null value.

Examples:

• The following examples show the values that are returned by the COMPARE_DECFLOAT function, given a variety of input decimal floating-point values:
COMPARE_DECFLOAT

COMPARE_DECFLOAT(DECFLOAT(2.17), DECFLOAT(2.17)) = 0
COMPARE_DECFLOAT(DECFLOAT(2.17), DECFLOAT(2.170)) = 2
COMPARE_DECFLOAT(DECFLOAT(2.170), DECFLOAT(2.17)) = 1
COMPARE_DECFLOAT(DECFLOAT(2.17), DECFLOAT(0.0)) = 2
COMPARE_DECFLOAT(INFINITY, INFINITY) = 0
COMPARE_DECFLOAT(INFINITY, -INFINITY) = 2
COMPARE_DECFLOAT(DECFLOAT(-2), INFINITY) = 1
COMPARE_DECFLOAT(NAN, NAN) = 3
COMPARE_DECFLOAT(DECFLOAT(-0.1), SNAN) = 3
The schema is SYSIBM.

The CONCAT function combines two arguments to form a string expression.

\( \text{expression1} \)

An expression that returns a value of any string data type, any numeric data type, or any datetime data type.

\( \text{expression2} \)

An expression that returns a value of any string data type, any numeric data type, or any datetime data type. However, some data types are not supported in combination with the data type of \( \text{expression1} \), as described below.

The arguments can be any combination of string (except binary string), numeric, and datetime values. When an argument is a non-string value, it is implicitly cast to VARCHAR. A binary string can only be concatenated with another binary string. However, through the castable process of function resolution, a binary string can be concatenated with a character string defined as FOR BIT DATA when the first argument is the binary string.

Concatenation involving both a character string argument and a graphic string argument is supported only in a Unicode database. The character argument is first converted to the graphic data type before the concatenation. Character strings defined as FOR BIT DATA cannot be cast to the graphic data type.

The result of the function is a string that consists of the first argument followed by the second argument. The data type and the length of the result is determined by the data types and lengths of the arguments, after any applicable casting is done. For more information, refer to the “Data Type and Length of Concatenated Operands” table in the “Expressions” topic.

If either argument can be null, the result can be null; if either argument is null, the result is the null value.

Notes

- No check is made for improperly formed mixed data when doing concatenation.
- The CONCAT function is identical to the CONCAT operator. For more information, see “Expressions”.

Example

- Concatenate the column FIRSTNME with the column LASTNAME.

  ```sql
  SELECT CONCAT(FIRSTNME, LASTNAME)
  FROM EMPLOYEE
  WHERE EMPNO = '000010'
  ```

  Returns the value “CHRISTINEHAAS”.

  Chapter 3. Functions  377
The schema is SYSIBM. (The SYSFUN version of the COS function continues to be available.)

Returns the cosine of the argument, where the argument is an angle expressed in radians.

The argument can be of any built-in numeric type (except for DECFLOAT). It is converted to a double-precision floating-point number for processing by the function.

The result of the function is a double-precision floating-point number. The result can be null if the argument can be null or the database is configured with `dft_sqlmathwarn` set to YES; the result is the null value if the argument is null.
The schema is SYSIBM.

Returns the hyperbolic cosine of the argument, where the argument is an angle expressed in radians.

The argument can be of any built-in numeric data type (except for DECFLOAT). It is converted to a double-precision floating-point number for processing by the function.

The result of the function is a double-precision floating-point number. The result can be null if the argument can be null or the database is configured with `dft_sqlmathwarn` set to YES; the result is the null value if the argument is null.
The schema is SYSIBM. (The SYSFUN version of the COT function continues to be available.)

Returns the cotangent of the argument, where the argument is an angle expressed in radians.

The argument can be of any built-in numeric type (except for DECFLOAT). It is converted to a double-precision floating-point number for processing by the function.

The result of the function is a double-precision floating-point number. The result can be null if the argument can be null or the database is configured with dft_sqlmathwarn set to YES; the result is the null value if the argument is null.
CURSOR_ROWCOUNT

CURSOR_ROWCOUNT

The schema is SYSIBM.

The CURSOR_ROWCOUNT function returns the cumulative count of all rows fetched by the specified cursor since the cursor was opened.

cursor-variable-name

The name of a SQL variable or SQL parameter of a cursor type. The underlying cursor of the cursor-variable-name must be open (SQLSTATE 24501).

The result is 0 if no FETCH action on the underlying cursor of the cursor-variable-name was performed prior to the evaluation of the function.

This function can only be used within a compound SQL (compiled) statement.

The data type of the result is BIGINT. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Examples

The following example shows how to use the function to retrieve the count of rows associated with the cursor curEmp and assign it to a variable named rows_fetched:

```
SET rows_fetched = CURSOR_ROWCOUNT(curEmp);
```
The schema is SYSIBM.

The DATAPARTITIONNUM function returns the sequence number (SYSDATAPARTITIONS.SEQNO) of the data partition in which the row resides. Data partitions are sorted by range, and sequence numbers start at 0. For example, the DATAPARTITIONNUM function returns 0 for a row that resides in the data partition with the lowest range.

The argument must be the qualified or unqualified name of any column in the table. Because row-level information is returned, the result is the same regardless of which column is specified. The column can have any data type.

If column-name references a column in a view, the expression for the column in the view must reference a column of the underlying base table, and the view must be deletable. A nested or common table expression follows the same rules as a view.

The data type of the result is INTEGER and is never null.

This function cannot be used as a source function when creating a user-defined function. Because the function accepts any data type as an argument, it is not necessary to create additional signatures to support user-defined distinct types.

The DATAPARTITIONNUM function cannot be used within check constraints or in the definition of generated columns (SQLSTATE 42881). The DATAPARTITIONNUM function cannot be used in a materialized query table (MQT) definition (SQLSTATE 428EC).

Examples

- **Example 1:** Retrieve the sequence number of the data partition in which the row for EMPLOYEE.EMPNO resides.

  ```sql
  SELECT DATAPARTITIONNUM (EMPNO)
  FROM EMPLOYEE
  ```

- **Example 2:** To convert a sequence number that is returned by DATAPARTITIONNUM (for example, 0) to a data partition name that can be used in other SQL statements (such as ALTER TABLE...DETACH PARTITION), you can query the SYSCAT.DATAPARTITIONS catalog view. Include the SEQNO obtained from DATAPARTITIONNUM in the WHERE clause, as shown in the following example.

  ```sql
  SELECT DATAPARTITIONNAME
  FROM SYSCAT.DATAPARTITIONS
  WHERE TABNAME = 'EMPLOYEE' AND SEQNO = 0
  ```

  results in the value 'PART0'.

---

**DATAPARTITIONNAME**

The schema is SYSIBM.

The DATAPARTITIONNAME function returns the name of the data partition in which the row resides. Data partitions are sorted by range, and sequence numbers start at 0. For example, the DATAPARTITIONNAME function returns the value 'PART0' for a row that resides in the data partition with the lowest range.

The argument must be the qualified or unqualified name of any column in the table. Because row-level information is returned, the result is the same regardless of which column is specified. The column can have any data type.

If column-name references a column in a view, the expression for the column in the view must reference a column of the underlying base table, and the view must be deletable. A nested or common table expression follows the same rules as a view.

The data type of the result is VARCHAR and is never null.

This function cannot be used as a source function when creating a user-defined function. Because the function accepts any data type as an argument, it is not necessary to create additional signatures to support user-defined distinct types.

The DATAPARTITIONNAME function cannot be used within check constraints or in the definition of generated columns (SQLSTATE 42881). The DATAPARTITIONNAME function cannot be used in a materialized query table (MQT) definition (SQLSTATE 428EC).

Examples

- **Example 1:** Retrieve the name of the data partition in which the row for EMPLOYEE.EMPNO resides.

  ```sql
  SELECT DATAPARTITIONNAME (EMPNO)
  FROM EMPLOYEE
  ```

- **Example 2:** To convert a sequence number that is returned by DATAPARTITIONNAME (for example, 0) to a data partition name that can be used in other SQL statements (such as ALTER TABLE...DETACH PARTITION), you can query the SYSCAT.DATAPARTITIONS catalog view. Include the SEQNO obtained from DATAPARTITIONNAME in the WHERE clause, as shown in the following example.

  ```sql
  SELECT DATAPARTITIONNAME
  FROM SYSCAT.DATAPARTITIONS
  WHERE TABNAME = 'EMPLOYEE' AND SEQNO = 0
  ```

  results in the value 'PART0'.
The schema is SYSIBM.

The DATE function returns a date from a value.

The argument must be a DATE, TIMESTAMP, a positive number less than or equal to 3,652,059, a valid string representation of a date or timestamp, or a string of length 7 that is not a CLOB or DBCLOB.

Only Unicode databases support an argument that is a graphic string representation of a date or a timestamp. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

If the argument is a string of length 7, it must represent a valid date in the form yyyy-mm-dd, where yyyy are digits denoting a year, and mm/dd are digits between 001 and 366, denoting a day of that year.

The result of the function is a DATE. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The other rules depend on the data type of the argument:
• If the argument is a DATE, TIMESTAMP, or valid string representation of a date or timestamp:
  – The result is the date part of the value.
• If the argument is a number:
  – The result is the date that is \( n - 1 \) days after January 1, 0001, where \( n \) is the integral part of the number.
• If the argument is a string with a length of 7:
  – The result is the date represented by the string.

Examples

Assume that the column RECEIVED (whose data type is TIMESTAMP) has an internal value equivalent to ‘1988-12-25T17:12:30.000000’.

• This example results in an internal representation of ‘1988-12-25’.
  \[ \text{DATE}(\text{RECEIVED}) \]
• This example results in an internal representation of ‘1988-12-25’.
  \[ \text{DATE}(‘1988-12-25’) \]
• This example results in an internal representation of ‘1988-12-25’.
  \[ \text{DATE}(‘1988-12-25’) \]
• This example results in an internal representation of ‘0001-02-04’.
  \[ \text{DATE}(35) \]
The schema is SYSIBM.

The **DAY** function returns the day part of a value.

The argument must be a **DATE**, **TIMESTAMP**, date duration, timestamp duration, or a valid character string representation of a date or timestamp that is not a CLOB. In a Unicode database, if a supplied argument is a graphic string (except DBCLOB), it is first converted to a character string before the function is executed.

The result of the function is a large integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The other rules depend on the data type of the argument:

- If the argument is a **DATE**, **TIMESTAMP**, or valid string representation of a date or timestamp:
  - The result is the day part of the value, which is an integer between 1 and 31.
- If the argument is a date duration or timestamp duration:
  - The result is the day part of the value, which is an integer between -99 and 99. A nonzero result has the same sign as the argument.

**Examples**

- Using the **PROJECT** table, set the host variable **END_DAY** (smallint) to the day that the WELD LINE PLANNING project (**PROJNAME**) is scheduled to stop (**PRENDATE**)..

  ```sql
  SELECT DAY(PRENDATE) INTO :END_DAY FROM PROJECT WHERE PROJNAME = 'WELD LINE PLANNING'
  ```

  Results in **END_DAY** being set to 15 when using the sample table.

- Assume that the column **DATE1** (whose data type is **DATE**) has an internal value equivalent to 2000-03-15 and the column **DATE2** (whose data type is **DATE**) has an internal value equivalent to 1999-12-31.

  ```sql
  SELECT DAY(DATE1 - DATE2)
  ```

  Results in the value 15.
The schema is SYSIBM. The SYSFUN version of the DAYNAME function continues to be available.

The DAYNAME function returns a character string containing the name of the day (for example, Friday) for the day portion of expression, based on locale-name or the value of the special register CURRENT LOCALE LC_TIME.

**expression**

An expression that returns a value of one of the following built-in data types: a DATE, TIMESTAMP, or a valid character string representation of a date or timestamp that is not a CLOB. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

**locale-name**

A character constant that specifies the locale used to determine the language of the result. The value of locale-name is not case-sensitive and must be a valid locale (SQLSTATE 42815). For information on valid locales and their naming, see “Locale names for SQL and XQuery”. If locale-name is not specified, the value of the special register CURRENT LOCALE LC_TIME is used.

The result is a varying-length character string. The length attribute is 100. If the resulting string exceeds the length attribute of the result, the result will be truncated. If the expression argument can be null, the result can be null; if the expression argument is null, the result is the null value. The code page of the result is the code page of the section.

**Notes**

- **Julian and Gregorian calendar**: The transition from the Julian calendar to the Gregorian calendar on 15 October 1582 is taken into account by this function. However, the SYSFUN version of the DAYNAME function assumes the Gregorian calendar for all calculations.

- **Determinism**: DAYNAME is a deterministic function. However, when locale-name is not explicitly specified, the invocation of the function depends on the value of the special register CURRENT LOCALE LC_TIME. This invocation that depends on the value of a special register cannot be used wherever special registers cannot be used (SQLSTATE 42621 or 428EC).

**Example**

- Assume that the variable TMSTAMP is defined as TIMESTAMP and has the following value: 2007–03–09–14.07.38.123456. The following examples show several invocations of the function and the resulting string values. The result type in each case is VARCHAR(100).

<table>
<thead>
<tr>
<th>Function invocation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>DAYNAME (TMSTAMP, 'CLDR 1.5:en_US')</td>
<td>Friday</td>
</tr>
<tr>
<td>DAYNAME (TMSTAMP, 'CLDR 1.5:de_DE')</td>
<td>Freitag</td>
</tr>
<tr>
<td>DAYNAME (TMSTAMP, 'CLDR 1.5:fr_FR')</td>
<td>vendredi</td>
</tr>
</tbody>
</table>
**DAYOFWEEK**

The schema is SYSFUN.

The DAYOFWEEK function returns the day of the week in the argument as an integer value in the range 1-7, where 1 represents Sunday.

The argument must be a DATE, TIMESTAMP, or a valid character string representation of a date or timestamp that is not a CLOB. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

The result of the function is INTEGER. The result can be null; if the argument is null, the result is the null value.
DAYOFWEEK_ISO

The schema is SYSFUN.

Returns the day of the week in the argument as an integer value in the range 1-7, where 1 represents Monday.

The argument must be a DATE, TIMESTAMP, or a valid character string representation of a date or timestamp that is not a CLOB. In a Unicode database, if a supplied argument is a graphic string (except DBCLOB), it is first converted to a character string before the function is executed.

The result of the function is INTEGER. The result can be null; if the argument is null, the result is the null value.
DAYOFYEAR

DAYOFYEAR

>>>DAYOFYEAR(<expression>)<<<

The schema is SYSFUN.

Returns the day of the year in the argument as an integer value in the range 1-366.

The argument must be a DATE, TIMESTAMP, or a valid character string representation of a date or timestamp that is not a CLOB. In a Unicode database, if a supplied argument is a graphic string (except DBCLOB), it is first converted to a character string before the function is executed.

The result of the function is INTEGER. The result can be null; if the argument is null, the result is the null value.
The schema is SYSIBM.

The DAYS function returns an integer representation of a date.

The argument must be a DATE, TIMESTAMP, or a valid character string representation of a date, or timestamp that is not a CLOB. In a Unicode database, if a supplied argument is a graphic string (except DBCLOB), it is first converted to a character string before the function is executed.

The result of the function is a large integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The result is 1 more than the number of days from January 1, 0001 to \( D \), where \( D \) is the date that would occur if the DATE function were applied to the argument.

**Examples**

- Using the PROJECT table, set the host variable EDUCATION_DAYS (int) to the number of elapsed days (PRENDATE - PRSTDATE) estimated for the project (PROJNO) 'IF2000'.

```
SELECT DAYS(PRENDATE) - DAYS(PRSTDATE)
  INTO :EDUCATION_DAYS
  FROM PROJECT
  WHERE PROJNO = 'IF2000'
```

Results in EDUCATION_DAYS being set to 396.

- Using the PROJECT table, set the host variable TOTAL_DAYS (int) to the sum of elapsed days (PRENDATE - PRSTDATE) estimated for all projects in department (DEPTNO) 'E21'.

```
SELECT SUM(DAYS(PRENDATE) - DAYS(PRSTDATE))
  INTO :TOTAL_DAYS
  FROM PROJECT
  WHERE DEPTNO = 'E21'
```

Results in TOTAL_DAYS being set to 1584 when using the sample table.
The schema is SYSIBM.

The DBCLOB function returns a DBCLOB representation of a graphic string type.

In a Unicode database, if a supplied argument is a character string, it is first converted to a graphic string before the function is executed. When the output string is truncated, such that the last character is a high surrogate, that surrogate is either:

- Left as is, if the supplied argument is a character string
- Converted to the blank character (X'0020'), if the supplied argument is a graphic string

Do not rely on these behaviors, because they might change in a future release.

The result of the function is a DBCLOB. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

**graphic-expression**

An expression that returns a value that is a graphic string.

**integer**

An integer value specifying the length attribute of the resulting DBCLOB data type. The value must be between 0 and 1,073,741,823. If integer is not specified, the length of the result is the same as the length of the first argument.
The schema is SYSIBM.

The DBPARTITIONNUM function returns the database partition number for a row. For example, if used in a SELECT clause, it returns the database partition number for each row in the result set.

The argument must be the qualified or unqualified name of any column in the table. Because row-level information is returned, the result is the same regardless of which column is specified. The column can have any data type.

If column-name references a column in a view, the expression for the column in the view must reference a column of the underlying base table, and the view must be deletable. A nested or common table expression follows the same rules as a view.

The specific row (and table) for which the database partition number is returned by the DBPARTITIONNUM function is determined from the context of the SQL statement that uses the function.

The database partition number returned on transition variables and tables is derived from the current transition values of the distribution key columns. For example, in a before insert trigger, the function returns the projected database partition number, given the current values of the new transition variables. However, the values of the distribution key columns might be modified by a subsequent before insert trigger. Thus, the final database partition number of the row when it is inserted into the database might differ from the projected value.

The data type of the result is INTEGER and is never null. If there is no db2nodes.cfg file, the result is 0.

This function cannot be used as a source function when creating a user-defined function. Because the function accepts any data type as an argument, it is not necessary to create additional signatures to support user-defined distinct types.

The DBPARTITIONNUM function cannot be used on replicated tables, within check constraints, or in the definition of generated columns (SQLSTATE 42881).

For compatibility with previous versions of DB2 products, NODENUMBER can be specified in place of DBPARTITIONNUM.

Examples:

- Count the number of instances in which the row for a given employee in the EMPLOYEE table is on a different database partition than the description of the employee’s department in the DEPARTMENT table.

  ```sql
  SELECT COUNT(*) FROM DEPARTMENT D, EMPLOYEE E
  WHERE D.DEPTNO=E.WORKDEPT
  AND DBPARTITIONNUM(E.LASTNAME) <> DBPARTITIONNUM(D.DEPTNO)
  ```

- Join the EMPLOYEE and DEPARTMENT tables so that the rows of the two tables are on the same database partition.

  ```sql
  SELECT * FROM DEPARTMENT D, EMPLOYEE E
  WHERE DBPARTITIONNUM(E.LASTNAME) = DBPARTITIONNUM(D.DEPTNO)
  ```
Using a before trigger on the EMPLOYEE table, log the employee number and the projected database partition number of any new row in the EMPLOYEE table in a table named EMPINSERTLOG1.

```sql
CREATE TRIGGER EMPINSLOGTRIG1
BEFORE INSERT ON EMPLOYEE
REFERENCING NEW AW NEWTABLE
FOR EACH ROW
INSERT INTO EMPINSERTLOG1
VALUES (NEWTABLE.EMPNO, DBPARTITIONNUM (NEWTABLE.EMPNO))
```
The DECFLOAT function returns a decimal floating-point representation of a number or a string representation of a number.

**numeric-expression**
An expression that returns a value of any built-in numeric data type.

**string-expression**
An expression that returns a value that is a character-string or Unicode graphic-string representation of a number with a length not greater than the maximum length of a character constant. The data type of string-expression must not be CLOB or DBCLOB (SQLSTATE 42884). Leading and trailing blanks are removed from the string. The resulting substring is folded to uppercase and must conform to the rules for forming an integer, decimal, floating-point, or decimal floating-point constant (SQLSTATE 22018) and not be greater than 42 bytes (SQLSTATE 42820).

34 or 16
Specifies the number of digits of precision for the result. The default is 34.

**decimal-character**
Specifies the single-byte character constant used to delimit the decimal digits in character-expression from the whole part of the number. The character cannot be a digit, plus (+), minus (-), or blank, and it can appear at most once in character-expression.

The result is the same number that would result from CAST(string-expression AS DECFLOAT(n)) or CAST(numeric-expression AS DECFLOAT(n)). Leading and trailing blanks are removed from the string.

The result of the function is a decimal floating-point number with the implicitly or explicitly specified number of digits of precision. If the first argument can be null, the result can be null; if the first argument is null, the result is the null value.

If necessary, the source is rounded to the precision of the target. The CURRENT DECFLOAT Rounding MODE special register determines the rounding mode.
DECFLOAT

Notes

- The CAST specification should be used to increase the portability of applications. For more information, see “CAST specification”.
- All numeric values are interpreted as integer, decimal, or floating-point constants and then cast to decimal floating-point. The use of a floating-point constant can result in round-off errors and is therefore strongly discouraged. Use the string to decimal floating-point version of the DECFLOAT function instead.

Examples

- Use the DECFLOAT function in order to force a DECFLOAT data type to be returned in a select-list for the EDLEVEL column (data type = SMALLINT) in the EMPLOYEE table. The EMPNO column should also appear in the select list.

```sql
SELECT EMPNO, DECFLOAT(EDLEVEL,16)
FROM EMPLOYEE
```
The schema is SYSIBM.

The DECFLOAT_FORMAT function returns a DECFLOAT(34) value that is based on the interpretation of the input string using the specified format.

**string-expression**

An expression that returns a value that is a CHAR and VARCHAR data type. In a Unicode database, if a supplied argument is a GRAPHIC or VARGRAPHIC data type, it is first converted to VARCHAR before evaluating the function. Leading and trailing blanks are removed from the string. If `format-string` is not specified, the resulting substring must conform to the rules for forming an SQL integer, decimal, floating-point, or decimal floating-point constant (SQLSTATE 22018) and not be greater than 42 bytes (SQLSTATE 42820); otherwise, the resulting substring must contain the components of a number that correspond to the format specified by `format-string` (SQLSTATE 22018).

**format-string**

An expression that returns a value that is a built-in character string data type (except CLOB). In a Unicode database, if a supplied argument is a graphic string (except DBCLOB), it is first converted to a character string before evaluating the function. The actual length must not be greater than 254 bytes (SQLSTATE 22018). The value is a template for how `string-expression` is to be interpreted for conversion to a DECFLOAT value. Format elements specified as a prefix can be used only at the beginning of the template. Format elements specified as a suffix can be used only at the end of the template. The format elements are case sensitive. The template must not contain more than one of the MI, S, or PR format elements (SQLSTATE 22018).

**Table 45. Format elements for the DECFLOAT_FORMAT function**

<table>
<thead>
<tr>
<th>Format element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 or 9</td>
<td>Each 0 or 9 represents a digit.</td>
</tr>
<tr>
<td>MI</td>
<td>Suffix: If <code>string-expression</code> is to represent a negative number, a trailing minus sign (−) is expected. If <code>string-expression</code> is to represent a positive number, a trailing blank can be specified.</td>
</tr>
<tr>
<td>S</td>
<td>Prefix: If <code>string-expression</code> is to represent a negative number, a leading minus sign (−) is expected. If <code>string-expression</code> is to represent a positive number, a leading plus sign (+) or leading blank can be specified.</td>
</tr>
<tr>
<td>PR</td>
<td>Suffix: If <code>string-expression</code> is to represent a negative number, a leading less than character (&lt;) and a trailing greater than character (&gt;) are expected. If <code>string-expression</code> is to represent a positive number, a leading space and a trailing space can be specified.</td>
</tr>
<tr>
<td>$</td>
<td>Prefix: A leading dollar sign ($) must be specified.</td>
</tr>
<tr>
<td>,</td>
<td>Specifies the expected location of a comma. This comma is used as a group separator.</td>
</tr>
<tr>
<td>.</td>
<td>Specifies the expected location of the period. This period is used as a decimal point.</td>
</tr>
</tbody>
</table>
If `format-string` is not specified, `string-expression` must conform to the rules for forming an SQL integer, decimal, floating-point, or decimal floating-point constant (SQLSTATE 22018) and have a length not greater than 42 bytes (SQLSTATE 42820).

The result is a DECFLOAT(34). If the first or second argument can be null, the result can be null; if the first or second argument is null, the result is the null value.

**Notes**
- Syntax alternatives: TO_NUMBER is a synonym for DECFLOAT_FORMAT.

**Examples**
- `DECFLOAT_FORMAT( '123.45' )`
  returns 123.45
- `DECFLOAT_FORMAT( '-123456.78' )`
  returns -123456.78
- `DECFLOAT_FORMAT( '+123456.78' )`
  returns 123456.78
- `DECFLOAT_FORMAT( '1.23E4' )`
  returns 12300
- `DECFLOAT_FORMAT( '123.4', '9999.99' )`
  returns 123.40
- `DECFLOAT_FORMAT( '001,234', '000,000' )`
  returns 1234
- `DECFLOAT_FORMAT( '1234 ', '9999MI' )`
  returns 1234
- `DECFLOAT_FORMAT( '1234−', '9999MI' )`
  returns -1234
- `DECFLOAT_FORMAT( '+1234', 'S9999' )`
  returns 1234
- `DECFLOAT_FORMAT( '-1234', 'S9999' )`
  returns -1234
- `DECFLOAT_FORMAT( ' 1234 ', '9999PR' )`
  returns 1234
- `DECFLOAT_FORMAT( '<1234>', '9999PR' )`
  returns -1234
- `DECFLOAT_FORMAT( '$123,456.78', '$999,999.99' )`
  returns 123456.78
DECIMAL or DEC

Numeric to Decimal:

\[
\text{DECIMAL}(\text{numeric-expression}, \text{precision}, \text{scale})
\]

String to Decimal:

\[
\text{DECIMAL}(\text{string-expression}, \text{precision}, \text{scale}, \text{decimal-character})
\]

Datetime to Decimal:

\[
\text{DECIMAL}(\text{datetime-expression}, \text{precision}, \text{scale})
\]

The schema is SYSIBM.

The DECIMAL function returns a decimal representation of:

- A number
- A string representation of a number
- A datetime value

**Numeric to Decimal**

- \textit{numeric-expression}
  
  An expression that returns a value of any built-in numeric data type.

- \textit{precision}
  
  An integer constant with a value in the range of 1 to 31.
  
  The default for \textit{precision} depends on the data type of \textit{numeric-expression}:
  
  - 31 for decimal floating-point
  - 15 for floating-point and decimal
  - 19 for big integer
  - 11 for large integer
  - 5 for small integer.

- \textit{scale}
  
  An integer constant in the range of 0 to the \textit{precision} value. The default is zero.

The result is the same number that would occur if the first argument were assigned to a decimal column or variable with a precision of \textit{precision} and a scale of \textit{scale}. An error is returned if the number of significant decimal digits required to represent the whole part of the number is greater than \textit{precision} - \textit{scale} (SQLSTATE 22003).
DECIMAL or DEC

String to Decimal

string-expression
An expression that returns a value that is a character string or a Unicode graphic-string representation of a number with a length not greater than the maximum length of a character constant. The data type of string-expression must not be CLOB or DBCLOB (SQLSTATE 42884). Leading and trailing blanks are eliminated from the string and the resulting string must conform to the rules for forming an integer, decimal, floating-point, or decimal floating-point constant (SQLSTATE 22018).

The string-expression is converted to the section code page if required to match the code page of the constant decimal-character.

precision
An integer constant with a value in the range 1 to 31 that specifies the precision of the result. If not specified, the default is 15.

scale
An integer constant with a value in the range 0 to precision that specifies the scale of the result. If not specified, the default is 0.

decimal-character
Specifies the single-byte character constant used to delimit the decimal digits in string-expression from the whole part of the number. The character cannot be a digit, plus (+), minus (-), or blank, and it can appear at most once in string-expression (SQLSTATE 42815).

The result is the same number that would result from CAST(string-expression AS DECIMAL(precision, scale)). Digits are truncated from the end of the decimal number if the number of digits to the right of the decimal separator character is greater than the scale scale. An error is returned if the number of significant digits to the left of the decimal character (the whole part of the number) in string-expression is greater than precision - scale (SQLSTATE 22003). The default decimal character is not valid in the substring if a different value for the decimal-character argument is specified (SQLSTATE 22018).

Datetime to Decimal

datetime-expression
An expression that returns a value of type DATE, TIME or TIMESTAMP.

precision
An integer constant with a value in the range 1 to 31 that specifies the precision of the result. If not specified, the default for the precision and scale depends on the data type of datetime-expression as follows:

- Precision is 8 and scale is 0 for a DATE. The result is a DECIMAL(8,0) value representing the date as yyyyymmdd.
- Precision is 6 and scale is 0 for a TIME. The result is a DECIMAL(6,0) value representing the time as hhmms.
- Precision is 14+tp and scale is tp for a TIMESTAMP(tp). The result is a DECIMAL(14+tp,tp) value representing the timestamp as yyyyymmddhhmms.sssssssssssssss.
An integer constant with a value in the range 0 to precision that specifies the scale of the result. If not specified and a precision is specified, the default is 0.

The result is the same number that would result from CAST(datetime - expression AS DECIMAL(precision, scale)). Digits are truncated from the end of the decimal number if the number of digits to the right of the decimal separator character is greater than the scale scale. An error is returned if the number of significant digits to the left of the decimal character (the whole part of the number) in string-expression is greater than precision - scale (SQLSTATE 22003).

If the first argument can be null, the result can be null; if the first argument is null, the result is the null value.

Note: The CAST specification should be used to increase the portability of applications. For more information, see “CAST specification”.

Examples
- Use the DECIMAL function in order to force a DECIMAL data type (with a precision of 5 and a scale of 2) to be returned in a select-list for the EDLEVEL column (data type = SMALLINT) in the EMPLOYEE table. The EMPNO column should also appear in the select list.
  ```sql
  SELECT EMPNO, DECIMAL(EDLEVEL,5,2)
  FROM EMPLOYEE
  ```
- Assume the host variable PERIOD is of type INTEGER. Then, in order to use its value as a date duration it must be "cast" as decimal(8,0).
  ```sql
  SELECT PRSTDATE + DECIMAL(:PERIOD,8)
  FROM PROJECT
  ```
- Assume that updates to the SALARY column are input through a window as a character string using comma as a decimal character (for example, the user inputs 21400,50). Once validated by the application, it is assigned to the host variable newsalary which is defined as CHAR(10).
  ```sql
  UPDATE STAFF
  SET SALARY = DECIMAL(:newsalary, 9, 2, ',')
  WHERE ID = :empid;
  ```
  The value of newsalary becomes 21400.50.
- Add the default decimal character (.) to a value.
  ```sql
  DECIMAL('21400,50', 9, 2, ',')
  ```
  This fails because a period (.) is specified as the decimal character, but a comma (,) appears in the first argument as a delimiter.
- Assume that the column STARTING (whose data type is TIME) has an internal value equivalent to '12:10:00'.
  ```sql
  DECIMAL(STARTING)
  ```
  results in the value 121 000.
- Assume that the column RECEIVED (whose data type is TIMESTAMP) has an internal value equivalent to '1988-12-22-14.07.21.136421'.
  ```sql
  DECIMAL(RECEIVED)
  ```
  results in the value 19 881 222 140 721.136421.
This example shows the decimal result and resulting precision and scale for various datetime input values. Assume the existence of the following columns with associated values:

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENDDT</td>
<td>DATE</td>
<td>2000-03-21</td>
</tr>
<tr>
<td>ENDTM</td>
<td>TIME</td>
<td>12:02:21</td>
</tr>
<tr>
<td>ENDTS</td>
<td>TIMESTAMP</td>
<td>2000-03-21-12.02.21.123456</td>
</tr>
<tr>
<td>ENDTS0</td>
<td>TIMESTAMP(0)</td>
<td>2000-03-21-12.02.21</td>
</tr>
<tr>
<td>ENDTS9</td>
<td>TIMESTAMP(9)</td>
<td>2000-03-21-12.02.21.123456789</td>
</tr>
</tbody>
</table>

The following table shows the decimal result and resulting precision and scale for various datetime input values.

<table>
<thead>
<tr>
<th>DECIMAL(arguments)</th>
<th>Precision and Scale</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>DECIMAL(ENDDT)</td>
<td>(8,0)</td>
<td>20000321.</td>
</tr>
<tr>
<td>DECIMAL(ENDDT, 10)</td>
<td>(10,0)</td>
<td>20000321.</td>
</tr>
<tr>
<td>DECIMAL(ENDDT, 12, 2)</td>
<td>(12,2)</td>
<td>20000321.00</td>
</tr>
<tr>
<td>DECIMAL(ENDTM)</td>
<td>(6,0)</td>
<td>120221.</td>
</tr>
<tr>
<td>DECIMAL(ENDTM, 10)</td>
<td>(10,0)</td>
<td>120221.</td>
</tr>
<tr>
<td>DECIMAL(ENDTM, 10, 2)</td>
<td>(10,2)</td>
<td>120221.00</td>
</tr>
<tr>
<td>DECIMAL(ENDTS)</td>
<td>(20,6)</td>
<td>20000321120221.123456</td>
</tr>
<tr>
<td>DECIMAL(ENDTS, 23)</td>
<td>(23,0)</td>
<td>20000321120221.123456</td>
</tr>
<tr>
<td>DECIMAL(ENDTS, 23, 4)</td>
<td>(23,4)</td>
<td>20000321120221.1234</td>
</tr>
<tr>
<td>DECIMAL(ENDTS0)</td>
<td>(14,0)</td>
<td>20000321120221.</td>
</tr>
<tr>
<td>DECIMAL(ENDTS9)</td>
<td>(23,9)</td>
<td>20000321120221.123456789</td>
</tr>
</tbody>
</table>
The schema is SYSIBM.

The DECODE function compares each expression2 to expression1. If expression1 is equal to expression2, or both expression1 and expression2 are null, the value of the following result-expression is returned. If no expression2 matches expression1, the value of else-expression is returned; otherwise a null value is returned.

The DECODE function is similar to the CASE expression except for the handling of null values:

- A null value of expression1 will match a corresponding null value of expression2.
- If the NULL keyword is used as an argument in the DECODE function, it must be cast to an appropriate data type.

The rules for determining the result type of a DECODE expression are based on the corresponding CASE expression.

Examples:

The DECODE expression:

```
DECODE (c1, 7, 'a', 6, 'b', 'c')
```

achieves the same result as the following CASE expression:

```
CASE c1
  WHEN 7 THEN 'a'
  WHEN 6 THEN 'b'
  ELSE 'c'
END
```

Similarly, the DECODE expression:

```
DECODE (c1, var1, 'a', var2, 'b')
```

where the values of c1, var1, and var2 could be null values, achieves the same result as the following CASE expression:

```
CASE
  WHEN c1 = var1 OR (c1 IS NULL AND var1 IS NULL) THEN 'a'
  WHEN c1 = var2 OR (c1 IS NULL AND var2 IS NULL) THEN 'b'
  ELSE NULL
END
```

Consider also the following query:

```
SELECT ID, DECODE (STATUS, 'A', 'Accepted', 'D', 'Denied',
                  CAST(NULL AS VARCHAR(1)), 'Unknown',
                  'Other')
FROM CONTRACTS
```

Here is the same statement using a CASE expression:

```
SELECT ID,
CASE
  WHEN STATUS = 'A' THEN 'Accepted'
```
WHEN STATUS = 'D' THEN 'Denied'
WHEN STATUS IS NULL THEN 'Unknown'
ELSE 'Other'
END
FROM CONTRACTS
The schema is SYSIBM.

The DECRYPT_BIN and DECRYPT_CHAR functions both return a value that is the result of decrypting encrypted-data. The password used for decryption is either the password-string-expression value or the encryption password value that was assigned by the SET ENCRYPTION PASSWORD statement. To maintain the best level of security on your system, it is recommended that you do not pass the encryption password explicitly with the DECRYPT_BIN and DECRYPT_CHAR functions in your query; instead, use the SET ENCRYPTION PASSWORD statement to set the password, and use a host variable or dynamic parameter markers when you use the SET ENCRYPTION PASSWORD statement, rather than a literal string.

The DECRYPT_BIN and DECRYPT_CHAR functions can only decrypt values that are encrypted using the ENCRYPT function (SQLSTATE 428FE).

encrypted-data
An expression that returns a CHAR FOR BIT DATA or VARCHAR FOR BIT DATA value as a complete, encrypted data string. The data string must have been encrypted using the ENCRYPT function.

password-string-expression
An expression that returns a CHAR or VARCHAR value with at least 6 bytes and no more than 127 bytes (SQLSTATE 428FC). This expression must be the same password used to encrypt the data (SQLSTATE 428FD). If the value of the password argument is null or not provided, the data will be decrypted using the encryption password value that was assigned for the session by the SET ENCRYPTION PASSWORD statement (SQLSTATE 51039).

The result of the DECRYPT_BIN function is VARCHAR FOR BIT DATA. The result of the DECRYPT_CHAR function is VARCHAR. If encrypted-data included a hint, the hint is not returned by the function. The length attribute of the result is the length of the data type of encrypted-data minus 8 bytes. The actual length of the value returned by the function will match the length of the original string that was encrypted. If encrypted-data includes bytes beyond the encrypted string, these bytes are not returned by the function.

If the first argument can be null, the result can be null. If the first argument is null, the result is the null value.

If the data is decrypted on a different system, which uses a code page that is different from the code page in which the data was encrypted, expansion might occur when converting the decrypted value to the database code page. In such situations, the encrypted-data value should be cast to a VARCHAR string with a larger number of bytes.

**Examples**

The following example demonstrates the use of the DECRYPT_CHAR function by showing code fragments from an embedded SQL application.
EXEC SQL BEGIN DECLARE SECTION;
  char hostVarCreateTableStmt[100];
  char hostVarSetEncPassStmt[200];
  char hostVarPassword[128];
  char hostVarInsertStmt1[200];
  char hostVarInsertStmt2[200];
  char hostVarSelectStmt1[200];
  char hostVarSelectStmt2[200];
EXEC SQL END DECLARE SECTION;

/* prepare the statement */
strcpy(hostVarCreateTableStmt, "CREATE TABLE EMP (SSN VARCHAR(24) FOR BIT DATA)");
EXEC SQL PREPARE hostVarCreateTableStmt FROM :hostVarCreateTableStmt;

/* execute the statement */
EXEC SQL EXECUTE hostVarCreateTableStmt;

Use the SET ENCRYPTION PASSWORD statement to set an encryption password for the session:
/* prepare the statement with a parameter marker */
strcpy(hostVarSetEncPassStmt, "SET ENCRYPTION PASSWORD = ?");
EXEC SQL PREPARE hostVarSetEncPassStmt FROM :hostVarSetEncPassStmt;

/* execute the statement for hostVarPassword = 'Pac1f1c' */
strcpy(hostVarPassword, "Pac1f1c");
EXEC SQL EXECUTE hostVarSetEncPassStmt USING :hostVarPassword;

/* prepare the statement */
strcpy(hostVarInsertStmt1, "INSERT INTO EMP(SSN) VALUES ENCRYPT('289-46-8832')");
EXEC SQL PREPARE hostVarInsertStmt1 FROM :hostVarInsertStmt1;

/* execute the statement */
EXEC SQL EXECUTE hostVarInsertStmt1;

/* prepare the statement */
strcpy(hostVarSelectStmt1, "SELECT DECRYPT_CHAR(SSN) FROM EMP");
EXEC SQL PREPARE hostVarSelectStmt1 FROM :hostVarSelectStmt1;

/* execute the statement */
EXEC SQL EXECUTE hostVarSelectStmt1;

This query returns the value '289-46-8832'.

Pass the encryption password explicitly:
/* prepare the statement */
strcpy(hostVarInsertStmt2, "INSERT INTO EMP (SSN) VALUES ENCRYPT('289-46-8832',?)");
EXEC SQL PREPARE hostVarInsertStmt2 FROM :hostVarInsertStmt2;

/* execute the statement for hostVarPassword = 'Pac1f1c' */
strcpy(hostVarPassword, "Pac1f1c");
EXEC SQL EXECUTE hostVarInsertStmt2 USING :hostVarPassword;

/* prepare the statement */
strcpy(hostVarSelectStmt2, "SELECT DECRYPT_CHAR(SSN,?) FROM EMP");
EXEC SQL PREPARE hostVarSelectStmt2 FROM :hostVarSelectStmt2;

/* execute the statement for hostVarPassword = 'Pac1f1c' */
strcpy(hostVarPassword, "Pac1f1c");
EXEC SQL EXECUTE hostVarSelectStmt2 USING :hostVarPassword;

This query returns the value '289-46-8832'.

DECRYPT_BIN and DECRYPT_CHAR
The schema is SYSIBM. (The SYSFUN version of the DEGREES function continues to be available.)

The DEGREES function returns the number of degrees of the argument, which is an angle expressed in radians.

The argument can be any built-in numeric data type. If the argument is decimal floating-point, the operation is performed in decimal floating-point; otherwise, the argument is converted to double-precision floating-point for processing by the function.

If the argument is DECFLOAT(n), the result is DECFLOAT(n); otherwise, the result is a double-precision floating-point number. The result can be null; if the argument is null, the result is the null value.

Example:
- Assume that RAD is a DECIMAL(4,3) host variable with a value of 3.142.
  
  VALUES DEGREES(:RAD)

  Returns the approximate value 180.0.
The DEREF function returns an instance of the target type of the argument.

The argument can be any value with a reference data type that has a defined scope (SQLSTATE 428DT).

The static data type of the result is the target type of the argument. The dynamic data type of the result is a subtype of the target type of the argument. The result can be null. The result is the null value if expression is a null value or if expression is a reference that has no matching OID in the target table.

The result is an instance of the subtype of the target type of the reference. The result is determined by finding the row of the target table or target view of the reference that has an object identifier that matches the reference value. The type of this row determines the dynamic type of the result. Since the type of the result can be based on a row of a subtable or subview of the target table or target view, the authorization ID of the statement must have SELECT privilege on the target table and all of its subtables or the target view and all of its subviews (SQLSTATE 42501).

Examples:

Assume that EMPLOYEE is a table of type EMP, and that its object identifier column is named EMPID. Then the following query returns an object of type EMP (or one of its subtypes), for each row of the EMPLOYEE table (and its subtables). This query requires SELECT privilege on EMPLOYEE and all its subtables.

```
SELECT DEREF(EMPID) FROM EMPLOYEE
```
DIFFERENCE

DIFFERENCE

The schema is SYSFUN.

Returns a value from 0 to 4 representing the difference between the sounds of two strings based on applying the SOUNDEX function to the strings. A value of 4 is the best possible sound match.

The arguments can be character strings that are either CHAR or VARCHAR not exceeding 4000 bytes. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed. The function interprets data that is passed to it as if it were ASCII characters, even if it is encoded in UTF-8.

The result of the function is INTEGER. The result can be null; if the argument is null, the result is the null value.

Example:

VALUES (DIFFERENCE('CONSTRAINT','CONSTANT'), SOUNDEX('CONSTRAINT'),
SOUNDEX('CONSTANT')),
(DIFFERENCE('CONSTRAINT','CONTRITE'), SOUNDEX('CONSTRAINT'),
SOUNDEX('CONTRITE'))

This example returns the following.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

----------- ---- ----
4 C523 C523
2 C523 C536

In the first row, the words have the same result from SOUNDEX while in the second row the words have only some similarity.
The schema is SYSIBM.

The DIGITS function returns a character-string representation of a number.

The expression must return a value that is a SMALLINT, INTEGER, BIGINT, DECIMAL, CHAR, or VARCHAR data type. In a Unicode database, if a supplied argument is a GRAPHIC or VARGRAPHIC data type, it is first converted to a character string before the function is executed. A CHAR or VARCHAR value is implicitly cast to DECIMAL(31,6) before evaluating the function.

If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The result of the function is a fixed-length character string representing the absolute value of the argument without regard to its scale. The result does not include a sign or a decimal character. Instead, it consists exclusively of digits, including, if necessary, leading zeros to fill out the string. The length of the string is:

- 5 if the argument is a small integer
- 10 if the argument is a large integer
- 19 if the argument is a big integer
- \( p \) if the argument is a decimal number with a precision of \( p \).

Examples:

- Assume that a table called TABLEX contains an INTEGER column called INTCOL containing 10-digit numbers. List all distinct four digit combinations of the first four digits contained in column INTCOL.

  ```sql
  SELECT DISTINCT SUBSTR(DIGITS(INTCOL),1,4)
  FROM TABLEX
  ```

- Assume that COLUMNX has the DECIMAL(6,2) data type, and that one of its values is -6.28. Then, for this value:

  ```sql
  DIGITS(COLUMNX)
  ```

  returns the value '000628'.

  The result is a string of length six (the precision of the column) with leading zeros padding the string out to this length. Neither sign nor decimal point appear in the result.
DOUBLEPRECISION or DOUBLE

Numeric to double:

```
DOUBLEPRECISION DOUBLE(numeric-expression)
```

Character string to double:

```
DOUBLEPRECISION DOUBLE(string-expression)
```

The schema is SYSIBM.

The DOUBLEPRECISION and DOUBLE functions return a double-precision floating-point representation of either:
- A number
- A string representation of a number

**Numeric to double**

`numeric-expression`

An expression that returns a value of any built-in numeric data type.

The result is the same number that would occur if the argument were assigned to a double-precision floating-point column or variable. If the numeric value of the argument is not within the range of double-precision floating-point, an error is returned (SQLSTATE 22003).

**Character string to double**

`string-expression`

An expression that returns a value that is character-string or Unicode graphic-string representation of a number. The data type of `string-expression` must not be CLOB (SQLSTATE 42884).

The result is the same number that would result from `CAST(string-expression AS DOUBLE PRECISION)`. Leading and trailing blanks are eliminated and the resulting string must conform to the rules for forming a valid numeric constant (SQLSTATE 22018). If the numeric value of the argument is not within the range of double-precision floating-point, an error is returned (SQLSTATE 22003).

The result of the function is a double-precision floating-point number. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

**Notes:**
- The CAST specification should be used to increase the portability of applications. For more information, see “CAST specification”.
- FLOAT is a synonym for DOUBLEPRECISION and DOUBLE.
- The SYSFUN version of DOUBLE (`string_expression`) continues to be available.
**Example:**

Using the EMPLOYEE table, find the ratio of salary to commission for employees whose commission is not zero. The columns involved (SALARY and COMM) have DECIMAL data types. To eliminate the possibility of out-of-range results, DOUBLE is applied to SALARY so that the division is carried out in floating point:

```sql
SELECT EMPNO, DOUBLE(SALARY)/COMM
FROM EMPLOYEE
WHERE COMM > 0
```
The schema is SYSIBM.

The empty value functions return a zero-length value of the associated data type. There are no arguments to these functions (the empty parentheses must be specified).

- The EMPTY_BLOB function returns a zero-length value with a data type of BLOB(1).
- The EMPTY_CLOB function returns a zero-length value with a data type of CLOB(1).
- The EMPTY_DBCLOB function returns a zero-length value with a data type of DBCLOB(1).

The result of these functions can be used in assignments to provide zero-length values where needed.

The schema is SYSIBM.
The schema is SYSIBM.

The ENCRYPT function returns a value that is the result of encrypting `data-string-expression`. The password used for encryption is either the `password-string-expression` value or the encryption password value that was assigned by the SET ENCRYPTION PASSWORD statement. To maintain the best level of security on your system, it is recommended that you do not pass the encryption password explicitly with the ENCRYPT function in your query; instead, use the SET ENCRYPTION PASSWORD statement to set the password, and use a host variable or dynamic parameter markers when you use the SET ENCRYPTION PASSWORD statement, rather than a literal string.

In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

`data-string-expression`

An expression that returns a CHAR or a VARCHAR value that is to be encrypted. The length attribute for the data type of `data-string-expression` is limited to 32663 without a `hint-string-expression` argument, and 32631 when the `hint-string-expression` argument is specified (SQLSTATE 42815).

`password-string-expression`

An expression that returns a CHAR or a VARCHAR value with at least 6 bytes and no more than 127 bytes (SQLSTATE 428FC). The value represents the password used to encrypt `data-string-expression`. If the value of the password argument is null or not provided, the data is encrypted using the encryption password value that was assigned for the session by the SET ENCRYPTION PASSWORD statement (SQLSTATE 51039).

`hint-string-expression`

An expression that returns a CHAR or a VARCHAR value with at most 32 bytes that will help data owners remember passwords (for example, 'Ocean' as a hint to remember 'Pacific'). If a hint value is given, the hint is embedded into the result and can be retrieved using the GETHINT function. If this argument is null or not provided, no hint will be embedded in the result.

The result data type of the function is VARCHAR FOR BIT DATA.

- When the optional hint parameter is specified, the length attribute of the result is equal to the length attribute of the unencrypted data + 8 bytes + the number of bytes until the next 8-byte boundary + 32 bytes for the length of the hint.
- When the optional hint parameter is not specified, the length attribute of the result is equal to the length attribute of the unencrypted data + 8 bytes + the number of bytes until the next 8-byte boundary.

If the first argument can be null, the result can be null. If the first argument is null, the result is the null value.
Note that the encrypted result is longer than the \textit{data-string-expression} value. Therefore, when assigning encrypted values, ensure that the target is declared with sufficient size to contain the entire encrypted value.

**Notes**

- **Encryption algorithm:** The internal encryption algorithm is RC2 block cipher with padding; the 128-bit secret key is derived from the password using an MD5 message digest.

- **Encryption passwords and data:** Password management is the user's responsibility. Once the data is encrypted, only the password that was used when encrypting it can be used to decrypt it (SQLSTATE 428FD).

The encrypted result might contain null terminator and other unprintable characters. Any assignment or cast to a length that is shorter than the suggested data length might result in failed decryption in the future, and lost data. Blanks are valid encrypted data values that might be truncated when stored in a column that is too short.

- **Administration of encrypted data:** Encrypted data can only be decrypted on servers that support the decryption functions corresponding to the ENCRYPT function. Therefore, replication of columns with encrypted data should only be done to servers that support the DECRYPT\_BIN or the DECRYPT\_CHAR function.

**Examples**

The following example demonstrates the use of the ENCRYPT function by showing code fragments from an embedded SQL application.

```sql
EXEC SQL BEGIN DECLARE SECTION;
   char hostVarCreateTableStmt[100];
   char hostVarSetEncPassStmt[200];
   char hostVarPassword[128];
   char hostVarInsertStmt1[200];
   char hostVarInsertStmt2[200];
   char hostVarInsertStmt3[200];
EXEC SQL END DECLARE SECTION;
/* prepare the statement */
strcpy(hostVarCreateTableStmt, "CREATE TABLE EMP (SSN VARCHAR(24) FOR BIT DATA)");
EXEC SQL PREPARE hostVarCreateTableStmt FROM :hostVarCreateTableStmt;
/* execute the statement */
EXEC SQL EXECUTE hostVarCreateTableStmt;

Use the SET ENCRYPTION PASSWORD statement to set an encryption password for the session:
/* prepare the statement with a parameter marker */
strcpy(hostVarSetEncPassStmt, "SET ENCRYPTION PASSWORD = ?");
EXEC SQL PREPARE hostVarSetEncPassStmt FROM :hostVarSetEncPassStmt;
/* execute the statement for hostVarPassword = 'Pac1f1c' */
strcpy(hostVarPassword, "Pac1f1c");
EXEC SQL EXECUTE hostVarSetEncPassStmt USING :hostVarPassword;
/* prepare the statement */
strcpy(hostVarInsertStmt1, "INSERT INTO EMP(SSN) VALUES ENCRYPT('289-46-8832')");
EXEC SQL PREPARE hostVarInsertStmt1 FROM :hostVarInsertStmt1;
/* execute the statement */
EXEC SQL EXECUTE hostVarInsertStmt1;
```

Chapter 3. Functions  413
Pass the encryption password explicitly:

```sql
EXEC SQL PREPARE hostVarInsertStmt2 FROM :hostVarInsertStmt2;

EXEC SQL EXECUTE hostVarInsertStmt2 USING :hostVarPassword;
```

Define a password hint:

```sql
EXEC SQL PREPARE hostVarInsertStmt3 FROM :hostVarInsertStmt3;

EXEC SQL EXECUTE hostVarInsertStmt3 USING :hostVarPassword;
```
The schema is SYSIBM.

The EVENT_MON_STATE function returns the current state of an event monitor.

The argument is a string expression with a resulting type of CHAR or VARCHAR and a value that is the name of an event monitor. If the named event monitor does not exist in the SYSCAT.EVENTMONITORS catalog table, SQLSTATE 42704 will be returned. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

The result is an integer with one of the following values:

0  The event monitor is inactive.
1  The event monitor is active.

If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Example:

The following example selects all of the defined event monitors, and indicates whether each is active or inactive:

```sql
SELECT EVMONNAME,
       CASE
           WHEN EVENT_MON_STATE(EVMONNAME) = 0 THEN 'Inactive'
           WHEN EVENT_MON_STATE(EVMONNAME) = 1 THEN 'Active'
       END
FROM SYSCAT.EVENTMONITORS
```
EXP

EXP

EXP(expression)

The schema is SYSIBM. (The SYSFUN version of the EXP function continues to be available.)

The EXP function returns a value that is the base of the natural logarithm (e) raised to a power specified by the argument. The EXP and LN functions are inverse operations.

The argument must be an expression that returns a value of any built-in numeric data type. If the argument is decimal floating-point, the operation is performed in decimal floating-point; otherwise, the argument is converted to double-precision floating-point for processing by the function.

If the argument is DECFLOAT(n), the result is DECFLOAT(n); otherwise, the result is a double-precision floating-point number. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Notes

Results involving DECFLOAT special values: For decimal floating-point values, the special values are treated as follows:
- EXP(NaN) returns NaN.
- EXP(-NaN) returns -NaN.
- EXP(Infinity) returns Infinity.
- EXP(-Infinity) returns 0.
- EXP(sNaN) returns NaN and a warning.
- EXP(-sNaN) returns -NaN and a warning.

Example
- Assume that E is a DECIMAL(10,9) host variable with a value of 3.453789832.
  VALUES EXP(:E)

  Returns the DOUBLE value +3.1620000069145E+001.
The schema is SYSIBM.

The EXTRACT function returns a portion of a date or timestamp based on its arguments.

If the argument can be null, the result can be null; if the argument is null, the result is the null value.

### Extract date values

**YEAR**

Specifies that the year portion of `date-expression` or `timestamp-expression` is returned. The result is identical to the `YEAR` scalar function.

**MONTH**

Specifies that the month portion of `date-expression` or `timestamp-expression` is returned. The result is identical to the `MONTH` scalar function.

**DAY**

Specifies that the day portion of `date-expression` or `timestamp-expression` is returned. The result is identical to the `DAY` scalar function.

**date-expression**

An expression that returns the value of either a built-in `DATE` or built-in character string data type.

If `date-expression` is a character or graphic string, it must be a valid string representation of a date that is not a CLOB. In a Unicode database, if `date-expression` is a graphic string, it is first converted to a character string before the function is executed.

**timestamp-expression**

An expression that returns the value of either a built-in `TIMESTAMP` or built-in character string data type.

If `timestamp-expression` is a character or graphic string, it must be a valid string representation of a timestamp that is not a CLOB. In a Unicode database, if `timestamp-expression` is a graphic string, it is first converted to a character string before the function is executed.

### Extract time values

**HOUR**

Specifies that the hour portion of `time-expression` or `timestamp-expression` is returned. The result is identical to the `HOUR` scalar function.
MINUTE
Specifies that the minute portion of time-expression or timestamp-expression is returned. The result is identical to the MINUTE scalar function.

SECOND
Specifies that the second portion of time-expression or timestamp-expression is returned. The result is identical to:
- SECOND(expression, 6) when the data type of expression is a TIME value or a string representation of a TIME or TIMESTAMP
- SECOND(expression, s) when the data type of expression is a TIMESTAMP(s) value

time-expression
An expression that returns the value of either a built-in TIME or built-in character string data type.

If time-expression is a character or graphic string, it must be a valid string representation of a time that is not a CLOB. In a Unicode database, if time-expression is a graphic string, it is first converted to a character string before the function is executed.

timestamp-expression
An expression that returns the value of a built-in DATE, TIMESTAMP or character string data type.

If timestamp-expression is a DATE, it is converted to a TIMESTAMP(0) value, assuming a time of exactly midnight (00.00.00).

If timestamp-expression is a character or graphic string, it must be a valid string representation of a timestamp or date that is not a CLOB. In a Unicode database, if timestamp-expression is a graphic string, it is first converted to a character string before the function is executed. The string is converted to a TIMESTAMP(6) value.

The data type of the result of the function depends on the part of the datetime value that is specified:
- If YEAR, MONTH, DAY, HOUR, or MINUTE is specified, the data type of the result is INTEGER.
- If SECOND is specified with a TIMESTAMP(p) value, the data type of the result is DECIMAL(2+p, p) where p is the fractional seconds precision.
- If SECOND is specified with a TIME value or a string representation of a TIME or TIMESTAMP, the data type of the result is DECIMAL(8,6).

Example
- Assume that the column PRSTDATE has an internal value that is equivalent to 1988-12-25:

  ```sql
  SELECT EXTRACT(MONTH FROM PRSTDATE)
  FROM PROJECT;
  ```
The schema is SYSIBM.

The FLOAT function returns a floating-point representation of a number. FLOAT is a synonym for DOUBLE.
The schema is SYSIBM. (The SYSFUN version of the FLOOR function continues to be available.)

Returns the largest integer value less than or equal to the argument.

The result of the function has the same data type and length attribute as the argument except that the scale is 0 if the argument is DECIMAL. For example, an argument with a data type of DECIMAL(5,5) returns DECIMAL(5,0).

The result can be null if the argument can be null or if the argument is not a decimal floating-point number and the database is configured with `dft_sqlmathwarn` set to YES; the result is the null value if the argument is null.

**Notes**

**Results involving DECFLOAT special values:** For decimal floating-point values, the special values are treated as follows:

- FLOOR(NaN) returns NaN.
- FLOOR(-NaN) returns -NaN.
- FLOOR(Infinity) returns Infinity.
- FLOOR(-Infinity) returns -Infinity.
- FLOOR(sNaN) returns NaN and a warning.
- FLOOR(-sNaN) returns -NaN and a warning.

**Examples**

- Use the FLOOR function to truncate any digits to the right of the decimal point.
  ```sql
  SELECT FLOOR(SALARY) 
  FROM EMPLOYEE
  ```

- Use the FLOOR function on both positive and negative numbers.
  ```sql
  VALUES FLOOR(3.5), FLOOR(3.1), 
  FLOOR(-3.1), FLOOR(-3.5)
  ```

  This example returns 3., 3., -4., and -4., respectively.
The schema is SYSIBM.

The GENERATE_UNIQUE function returns a bit data character string 13 bytes long (CHAR(13) FOR BIT DATA) that is unique compared to any other execution of the same function. (The system clock is used to generate the internal Universal Time, Coordinated (UTC) timestamp along with the database partition number on which the function executes. Adjustments that move the actual system clock backward could result in duplicate values.) The function is defined as non-deterministic.

There are no arguments to this function (the empty parentheses must be specified).

The result of the function is a unique value that includes the internal form of the Universal Time, Coordinated (UTC) and the database partition number where the function was processed. The result cannot be null.

The result of this function can be used to provide unique values in a table. Each successive value will be greater than the previous value, providing a sequence that can be used within a table. The value includes the database partition number where the function executed so that a table partitioned across multiple database partitions also has unique values in some sequence. The sequence is based on the time the function was executed.

This function differs from using the special register CURRENT TIMESTAMP in that a unique value is generated for each row of a multiple row insert statement or an insert statement with a fullselect.

The timestamp value that is part of the result of this function can be determined using the TIMESTAMP scalar function with the result of GENERATE_UNIQUE as an argument.

Examples:
• Create a table that includes a column that is unique for each row. Populate this column using the GENERATE_UNIQUE function. Notice that the UNIQUE_ID column has "FOR BIT DATA" specified to identify the column as a bit data character string.

```
CREATE TABLE EMP_UPDATE
  (UNIQUE_ID CHAR(13) FOR BIT DATA,
   EMPNO CHAR(6),
   TEXT VARCHAR(1000))
INSERT INTO EMP_UPDATE
VALUES (GENERATE_UNIQUE(), '000020', 'Update entry...'),
       (GENERATE_UNIQUE(), '000050', 'Update entry...')
```

This table will have a unique identifier for each row provided that the UNIQUE_ID column is always set using GENERATE_UNIQUE. This can be done by introducing a trigger on the table.

```
CREATE TRIGGER EMP_UPDATE_UNIQUE
  NO CASCADE BEFORE INSERT ON EMP_UPDATE
  REFERENCING NEW AS NEW_UPD
  FOR EACH ROW
  WHEN (NEW_UPD.UNIQUE_ID = GENERATE_UNIQUE())
```

Chapter 3. Functions  421
With this trigger defined, the previous INSERT statement could be issued without the first column as follows.

```sql
INSERT INTO EMP_UPDATE (EMPNO, TEXT)
VALUES ('000020', 'Update entry 1...'),
       ('000050', 'Update entry 2...')
```

The timestamp (in UTC) for when a row was added to EMP_UPDATE can be returned using:

```sql
SELECT TIMESTAMP (UNIQUE_ID), EMPNO, TEXT
FROM EMP_UPDATE
```

Therefore, there is no need to have a timestamp column in the table to record when a row is inserted.
The schema is SYSIBM.

The GETHINT function will return the password hint if one is found in the encrypted-data. A password hint is a phrase that will help data owners remember passwords; for example, ‘Ocean’ as a hint to remember ‘Pacific’. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

encrypted-data
   An expression that returns a CHAR FOR BIT DATA or VARCHAR FOR BIT DATA value that is a complete, encrypted data string. The data string must have been encrypted using the ENCRYPT function (SQLSTATE 428FE).

The result of the function is VARCHAR(32). The result can be null; if the hint parameter was not added to the encrypted-data by the ENCRYPT function or the first argument is null, the result is the null value.

Example:

In this example the hint ‘Ocean’ is stored to help the user remember the encryption password ‘Pacific’.

```
INSERT INTO EMP (SSN) VALUES ENCRYPT('289-46-8832', 'Pacific','Ocean');
SELECT GETHINT(SSN)
FROM EMP;
```

The value returned is ‘Ocean’.
The schema is SYSIBM. The function name cannot be specified as a qualified name when keywords are used in the function signature.

The GRAPHIC function returns a fixed-length graphic string representation of:

- An integer number (Unicode database only), if the first argument is a SMALLINT, INTEGER, or BIGINT
- A decimal number (Unicode database only), if the first argument is a decimal number
• A double-precision floating-point number (Unicode database only), if the first argument is a DOUBLE or REAL
• A decimal floating-point number (Unicode database only), if the argument is a decimal floating-point number (DECFLOAT)
• A character string, converting single-byte characters to double-byte characters, if the first argument is any type of character string
• A graphic string, if the first argument is any type of graphic string
• A datetime value (Unicode database only), if the first argument is a DATE, TIME, or TIMESTAMP

In a Unicode database, if a supplied argument is a character string, it is first converted to a graphic string before the function is executed. When the output string is truncated, such that the last character is a high surrogate, that surrogate is converted to the blank character (X'0020'). Do not rely on this behavior, because it might change in a future release.

The result of the function is a fixed-length graphic string. If the first argument can be null, the result can be null; if the first argument is null, the result is the null value.

**Integer to graphic**

\[
\text{integer-expression}
\]

An expression that returns a value that is of an integer data type (SMALLINT, INTEGER, or BIGINT).

The result is a fixed-length graphic string representation of \( \text{integer-expression} \) in the form of an SQL integer constant. The result consists of \( n \) double-byte characters, which represent the significant digits in the argument, and is preceded by a minus sign if the argument is negative. The result is left justified.

• If the first argument is a small integer, the length of the result is 6.
• If the first argument is a large integer, the length of the result is 11.
• If the first argument is a big integer, the length of the result is 20.

If the number of double-byte characters in the result is less than the defined length of the result, the result is padded on the right with blanks.

The code page of the result is the DBCS code page of the section.

**Decimal to graphic**

\[
\text{decimal-expression}
\]

An expression that returns a value that is a decimal data type. The DECIMAL scalar function can be used to change the precision and scale.

\[
\text{decimal-character}
\]

Specifies the double-byte character constant that is used to delimit the decimal digits in the result graphic string. The double-byte character constant cannot be a digit, the plus sign (+), the minus sign (−), or a blank (SQLSTATE 42815). The default is the period (.) character.

The result is a fixed-length graphic string representation of \( \text{decimal-expression} \) in the form of an SQL decimal constant. The length of the result is \( 2+p \), where \( p \) is the precision of \( \text{decimal-expression} \). Leading zeros are not included. Trailing zeros are included. If \( \text{decimal-expression} \) is negative, the first double-byte character of the result is a minus sign;
otherwise, the first double-byte character is a digit or the decimal character. If the scale of decimal-expression is zero, the decimal character is not returned. If the number of double-byte characters in the result is less than the defined length of the result, the result is padded on the right with blanks.

The code page of the result is the DBCS code page of the section.

**Floating-point to graphic**

\[ \text{floating-point-expression} \]

An expression that returns a value that is a floating-point data type (DOUBLE or REAL).

\[ \text{decimal-character} \]

Specifies the double-byte character constant that is used to delimit the decimal digits in the result graphic string. The double-byte character constant cannot be a digit, the plus sign (+), the minus sign (-), or a blank (SQLSTATE 42815). The default is the period (.) character.

The result is a fixed-length graphic string representation of floating-point-expression in the form of an SQL floating-point constant. The length of the result is 24. The result is the smallest number of double-byte characters that can represent the value of floating-point-expression such that the mantissa consists of a single digit other than zero followed by a period and a sequence of digits. If floating-point-expression is negative, the first double-byte character of the result is a minus sign; otherwise, the first double-byte character is a digit. If floating-point-expression is zero, the result is 0E0. If the number of double-byte characters in the result is less than 24, the result is padded on the right with blanks.

The code page of the result is the DBCS code page of the section.

**Decimal floating-point to graphic**

\[ \text{decimal-floating-point-expression} \]

An expression that returns a value that is a decimal floating-point data type (DECFLOAT).

\[ \text{decimal-character} \]

Specifies the double-byte character constant that is used to delimit the decimal digits in the result graphic string. The double-byte character constant cannot be a digit, the plus sign (+), the minus sign (-), or a blank (SQLSTATE 42815). The default is the period (.) character.

The result is a fixed-length graphic string representation of decimal-floating-point-expression in the form of an SQL decimal floating-point constant. The length attribute of the result is 42. The result is the smallest number of double-byte characters that can represent the value of decimal-floating-point-expression. If decimal-floating-point-expression is negative, the first double-byte character of the result is a minus sign; otherwise, the first double-byte character is a digit. If decimal-floating-point-expression is zero, the result is 0.

If the value of decimal-floating-point-expression is the special value Infinity, sNaN, or NaN, the strings G’INFINITY’, G’SNaN’, and G’NaN’, respectively, are returned. If the special value is negative, the first double-byte character of the result is a minus sign. The decimal floating-point special value sNaN does not result in a warning when
converted to a string. If the number of double-byte characters in the result is less than 42, the result is padded on the right with blanks.

The code page of the result is the DBCS code page of the section.

**Character to graphic**

*character-expression*

An expression that returns a value that is a built-in character string data type (CHAR, VARCHAR, or CLOB).

*integer*

The length attribute of the resulting fixed-length graphic string. The value must be between 0 and 127. If the second argument is not specified:

- If the *character-expression* is the empty string constant, the length attribute of the result is 0.
- Otherwise, the length attribute of the result is the same as the length attribute of the first argument. If the actual length of the first argument (including trailing blanks) is greater than 127, an error is returned (SQLSTATE 22001).

The actual length of the result is the same as the length attribute of the result. If the length of the *character-expression* is less than the length of the result, the result is padded with blanks up to the length of the result. If the length of the *character-expression* is greater than the length attribute of the result, truncation is performed with no warning returned.

**Graphic to graphic**

*graphic-expression*

An expression that returns a built-in value that is a graphic string data type (GRAPHIC, VARGRAPHIC, or DBCLOB).

*integer*

The length attribute for the resulting fixed-length graphic string. The value must be between 0 and 127.

If the second argument is not specified:

- If the *graphic-expression* is the empty string constant, the length attribute of the result is 0.
- Otherwise, the length attribute of the result is the same as the length attribute of the first argument. If the actual length of the first argument (excluding trailing blanks) is greater than 127, an error is returned (SQLSTATE 22001).

The actual length of the result is the same as the length attribute of the result. If the length of the *graphic-expression* is less than the length of the result, the result is padded with blanks up to the length of the result. If the length of the *graphic-expression* is greater than the length attribute of the result, truncation is performed. A warning is returned (SQLSTATE 01004) unless the truncated characters were all blanks and the *graphic-expression* was not a DBCLOB.

**Datetime to graphic**

*datetime-expression*

An expression that is of one of the following data types:

- **DATE** The result is the graphic string representation of the date in the
format specified by the second argument. The length of the result is 10. An error is returned if the second argument is specified and is not a valid value (SQLSTATE 42703).

**TIME** The result is the graphic string representation of the time in the format specified by the second argument. The length of the result is 8. An error is returned if the second argument is specified and is not a valid value (SQLSTATE 42703).

**TIMESTAMP**

The result is the graphic string representation of the timestamp. If the data type of `datetime-expression` is `TIMESTAMP(0)`, the length of the result is 19. If the data type of `datetime-expression` is `TIMESTAMP(n)`, where \( n \) is between 1 and 12, the length of the result is \( 20+n \). Otherwise, the length of the result is 26.

The code page of the string is the code page of the section.

**Note:** The CAST specification should be used to increase the portability of applications when the first argument is numeric, or if the first argument is a string and the length argument is specified. For more information, see “CAST specification”.

**Examples**

- The EDLEVEL column is defined as SMALLINT. The following returns the value as a fixed-length graphic string.

```
SELECT GRAPHIC(EDLEVEL)
FROM EMPLOYEE
WHERE LASTNAME = 'HAAS'
```

Results in the value G’18 ’.

- The SALARY and COMM columns are defined as DECIMAL with a precision of 9 and a scale of 2. Return the total income for employee Haas using the comma decimal character.

```
SELECT GRAPHIC(SALARY + COMM, ',',)
FROM EMPLOYEE
WHERE LASTNAME = 'HAAS'
```

Results in the value G’56970,00 ’.
The schema is SYSIBM.

The GREATEST function returns the maximum value in a set of values.

The arguments must be compatible and each argument must be an expression that returns a value of any data type other than ARRAY, LOB, XML, a distinct type based on any of these types, or a structured type (SQLSTATE 42815). This function cannot be used as a source function when creating a user-defined function. Because this function accepts any compatible data types as arguments, it is not necessary to create additional signatures to support user-defined distinct types.

The selected argument is converted, if necessary, to the attributes of the result. The attributes of the result are determined by all the operands based on the rules for result data types.

The result of the function is the largest argument value. If at least one argument can be null, the result can be null; if either argument is null, the result is the null value.

The GREATEST scalar function is a synonym for the MAX scalar function.

Examples:

Assume that table T1 contains three columns C1, C2, and C3 with values 1, 7, and 4, respectively. The query:

```
SELECT GREATEST (C1, C2, C3) FROM T1
```

returns 7.

If column C3 has a value of NULL instead of 4, the same query returns NULL.
The schema is SYSIBM.

The HASHEDVALUE function returns the distribution map index of the row obtained by applying the partitioning function on the distribution key value of the row. For example, if used in a SELECT clause, it returns the distribution map index for each row of the table that was used to form the result of the SELECT statement.

The distribution map index returned on transition variables and tables is derived from the current transition values of the distribution key columns. For example, in a before insert trigger, the function will return the projected distribution map index given the current values of the new transition variables. However, the values of the distribution key columns may be modified by a subsequent before insert trigger. Thus, the final distribution map index of the row when it is inserted into the database may differ from the projected value.

The argument must be the qualified or unqualified name of a column in a table. The column can have any data type. (This function cannot be used as a source function when creating a user-defined function. Because it accepts any data type as an argument, it is not necessary to create additional signatures to support user-defined distinct types.) If column-name references a column of a view the expression in the view for the column must reference a column of the underlying base table and the view must be deletable. A nested or common table expression follows the same rules as a view.

The specific row (and table) for which the distribution map index is returned by the HASHEDVALUE function is determined from the context of the SQL statement that uses the function.

The data type of the result is INTEGER in the range 0 to 32767. For a table with no distribution key, the result is always 0. A null value is never returned. Since row-level information is returned, the results are the same, regardless of which column is specified for the table.

The HASHEDVALUE function cannot be used on replicated tables, within check constraints, or in the definition of generated columns (SQLSTATE 42881).

For compatibility with versions earlier than Version 8, the function name PARTITION can be substituted for HASHEDVALUE.

Example:

- List the employee numbers (EMPNO) from the EMPLOYEE table for all rows with a distribution map index of 100.

```
SELECT EMPNO FROM EMPLOYEE
WHERE HASHEDVALUE(PHONENO) = 100
```

- Log the employee number and the projected distribution map index of the new row into a table called EMPINSERTLOG2 for any insertion of employees by creating a before trigger on the table EMPLOYEE.
CREATE TRIGGER EMPINSLOGTRIG2
BEFORE INSERT ON EMPLOYEE
REFERENCING NEW AW NEWTABLE
FOR EACH ROW
INSERT INTO EMPINSERTLOG2
  VALUES(NEWTABLE.EMPNO, HASHEDVALUE(NEWTABLE.EMPNO))
The schema is SYSIBM.

The HEX function returns a hexadecimal representation of a value as a character string.

The argument can be an expression that is a value of any built-in data type with a maximum length of 16,336 bytes.

The result of the function is a character string. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The code page is the section code page.

The result is a string of hexadecimal digits. The first two represent the first byte of the argument, the next two represent the second byte of the argument, and so forth. If the argument is a datetime value or a numeric value the result is the hexadecimal representation of the internal form of the argument. The hexadecimal representation that is returned may be different depending on the application server where the function is executed. Cases where differences would be evident include:

- Character string arguments when the HEX function is performed on an ASCII client with an EBCDIC server or on an EBCDIC client with an ASCII server.
- Numeric arguments (in some cases) when the HEX function is performed where client and server systems have different byte orderings for numeric values.

The type and length of the result vary based on the type and length of character string arguments.

- Character string
  - Fixed length not greater than 127
    - Result is a character string of fixed length twice the defined length of the argument.
  - Fixed length greater than 127
    - Result is a character string of varying length twice the defined length of the argument.
  - Varying length
    - Result is a character string of varying length with maximum length twice the defined maximum length of the argument.

- Graphic string
  - Fixed length not greater than 63
    - Result is a character string of fixed length four times the defined length of the argument.
  - Fixed length greater than 63
    - Result is a character string of varying length four times the defined length of the argument.
  - Varying length
    - Result is a character string of varying length with maximum length four times the defined maximum length of the argument.
HEX

Examples:

Assume the use of a DB2 for AIX application server for the following examples.

- Using the DEPARTMENT table set the host variable HEX_MGRNO (char(12)) to the hexadecimal representation of the manager number (MGRNO) for the 'PLANNING' department (DEPTNAME).

```sql
SELECT HEX(MGRNO)
INTO :HEX_MGRNO
FROM DEPARTMENT
WHERE DEPTNAME = 'PLANNING'
```

HEX_MGRNO will be set to '303030303230' when using the sample table (character value is '000020').

- Suppose COL_1 is a column with a data type of char(1) and a value of 'B'. The hexadecimal representation of the letter 'B' is X'42'. HEX(COL_1) returns a two byte long string '42'.

- Suppose COL_3 is a column with a data type of decimal(6,2) and a value of 40.1. An eight byte long string '0004010C' is the result of applying the HEX function to the internal representation of the decimal value, 40.1.
The schema is SYSIBM.

The HOUR function returns the hour part of a value.

An expression that returns a value of one of the following built-in data types: a DATE, a TIME, a TIMESTAMP, a character string, or an exact numeric data type.

- If expression is a character string, it must not be a CLOB and its value must be a valid string representation of a datetime value. For the valid formats of string representations of datetime values, see “String representations of datetime values” in “Datetime values”.
- If expression is an exact numeric value, it must be a time duration or timestamp duration. For information about valid time durations and timestamp durations, see “Datet ime operands and durations”.
- Only Unicode databases support an expression that is a valid graphic string representation of a datetime value that is not a DBCLOB. The graphic string is converted to a character string before the function is executed.

The result of the function is a large integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The other rules depend on the data type of the argument:

- If the argument is a TIME, TIMESTAMP or valid string representation of a time or timestamp:
  - The result is the hour part of the value, which is an integer between 0 and 24.
- If the argument is a DATE or valid string representation of a date:
  - The result is 0.
- If the argument is a time duration or timestamp duration:
  - The result is the hour part of the value, which is an integer between -99 and 99. A nonzero result has the same sign as the argument.

Example

Using the CL_SCHED sample table, select all the classes that start in the afternoon.

```sql
SELECT * FROM CL_SCHED
WHERE HOUR(STARTING) BETWEEN 12 AND 17
```
IDENTITY_VAL_LOCAL

The schema is SYSIBM.

The IDENTITY_VAL_LOCAL function is a non-deterministic function that returns the most recently assigned value for an identity column, where the assignment occurred as a result of a single INSERT statement using a VALUES clause. The function has no input parameters.

The result is a DECIMAL(31,0), regardless of the actual data type of the corresponding identity column.

The value returned by the function is the value assigned to the identity column of the table identified in the most recent single row insert operation. The INSERT statement must contain a VALUES clause on a table containing an identity column. The INSERT statement must also be issued at the same level; that is, the value must be available locally at the level it was assigned, until it is replaced by the next assigned value. (A new level is initiated each time a trigger or routine is invoked.)

The assigned value is either a value supplied by the user (if the identity column is defined as GENERATED BY DEFAULT), or an identity value generated by the database manager.

It is recommended that a SELECT FROM data-change-table-reference statement be used to obtain the assigned value for an identity column. See "table-reference" in "subselect" for more information.

The function returns a null value if a single row INSERT statement with a VALUES clause has not been issued at the current processing level against a table containing an identity column.

The result of the function is not affected by the following:
  • A single row INSERT statement with a VALUES clause for a table without an identity column
  • A multiple row INSERT statement with a VALUES clause
  • An INSERT statement with a fullselect
  • A ROLLBACK TO SAVEPOINT statement

Notes
  • Expressions in the VALUES clause of an INSERT statement are evaluated prior to the assignments for the target columns of the insert operation. Thus, an invocation of an IDENTITY_VAL_LOCAL function inside the VALUES clause of an INSERT statement will use the most recently assigned value for an identity column from a previous insert operation. The function returns the null value if no previous single row INSERT statement with a VALUES clause for a table containing an identity column has been executed within the same level as the IDENTITY_VAL_LOCAL function.
  • The identity column value of the table for which the trigger is defined can be determined within a trigger by referencing the trigger transition variable for the identity column.
IDENTITY_VAL_LOCAL

- The result of invoking the IDENTITY_VAL_LOCAL function from within the trigger condition of an insert trigger is a null value.

- It is possible that multiple before or after insert triggers exist for a table. In this case, each trigger is processed separately, and identity values assigned by one triggered action are not available to other triggered actions using the IDENTITY_VAL_LOCAL function. This is true even though the multiple triggered actions are conceptually defined at the same level.

- It is not generally recommended to use the IDENTITY_VAL_LOCAL function in the body of a before insert trigger. The result of invoking the IDENTITY_VAL_LOCAL function from within the triggered action of a before insert trigger is the null value. The value for the identity column of the table for which the trigger is defined cannot be obtained by invoking the IDENTITY_VAL_LOCAL function within the triggered action of a before insert trigger. However, the value for the identity column can be obtained in the triggered action by referencing the trigger transition variable for the identity column.

- The result of invoking the IDENTITY_VAL_LOCAL function from within the triggered action of an after insert trigger is the value assigned to an identity column of the table identified in the most recent single row insert operation invoked in the same triggered action that had a VALUES clause for a table containing an identity column. (This applies to both FOR EACH ROW and FOR EACH STATEMENT after insert triggers.) If a single row INSERT statement with a VALUES clause for a table containing an identity column was not executed within the same triggered action, prior to the invocation of the IDENTITY_VAL_LOCAL function, the function returns a null value.

- Because IDENTITY_VAL_LOCAL is a non-deterministic function, the result of invoking this function within the SELECT statement of a cursor can vary for each FETCH statement.

- The assigned value is the value actually assigned to the identity column (that is, the value that would be returned on a subsequent SELECT statement). This value is not necessarily the value provided in the VALUES clause of the INSERT statement, or a value generated by the database manager. The assigned value could be a value specified in a SET transition variable statement, within the body of a before insert trigger, for a trigger transition variable associated with the identity column.

- **Scope of IDENTITY_VAL_LOCAL:** The IDENTITY_VAL_LOCAL value persists until the next insert in the current session into a table that has an identity column defined on it, or the application session ends. The value is unaffected by COMMIT or ROLLBACK statements. The IDENTITY_VAL_LOCAL value cannot be directly set and is a result of inserting a row into a table.

  A technique commonly used, especially for performance, is for an application or product to manage a set of connections and route transactions to an arbitrary connection. In these situations, the availability of the IDENTITY_VAL_LOCAL value should be relied on only until the end of the transaction. Examples of where this type of situation can occur include applications that use XA protocols, use connection pooling, use the connection concentrator, and use HADR to achieve failover.

  The value returned by the function following a failed single row INSERT statement with a VALUES clause into a table with an identity column is unpredictable. It could be the value that would have been returned from the function had it been invoked prior to the failed insert operation, or it could be the value that would have been assigned had the insert operation succeeded. The actual value returned depends on the point of failure, and is therefore unpredictable.
IDENTITY_VAL_LOCAL

Examples:

Example 1: Create two tables, T1 and T2, each with an identity column named C1. Start the identity sequence for table T2 at 10. Insert some values for C2 into T1.

```
CREATE TABLE T1
(C1 INTEGER GENERATED ALWAYS AS IDENTITY,
 C2 INTEGER)

CREATE TABLE T2
(C1 DECIMAL(15,0) GENERATED BY DEFAULT AS IDENTITY (START WITH 10),
 C2 INTEGER)

INSERT INTO T1 (C2) VALUES (5)
INSERT INTO T1 (C2) VALUES (6)

SELECT * FROM T1
```

This query returns:

<table>
<thead>
<tr>
<th>C1</th>
<th>C2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
</tr>
</tbody>
</table>

Insert a single row into table T2, where column C2 gets its value from the IDENTITY_VAL_LOCAL function.

```
INSERT INTO T2 (C2) VALUES (IDENTITY_VAL_LOCAL())

SELECT * FROM T2
```

This query returns:

<table>
<thead>
<tr>
<th>C1</th>
<th>C2</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.</td>
<td>2</td>
</tr>
</tbody>
</table>

Example 2: In a nested environment involving a trigger, use the IDENTITY_VAL_LOCAL function to retrieve the identity value assigned at a particular level, even though there might have been identity values assigned at lower levels. Assume that there are three tables, EMPLOYEE, EMP_ACT, and ACCT_LOG. There is an after insert trigger defined on EMPLOYEE that results in additional inserts into the EMP_ACT and ACCT_LOG tables.

```
CREATE TABLE EMPLOYEE
(EMPNO SMALLINT GENERATED ALWAYS AS IDENTITY (START WITH 1000),
 NAME CHAR(30),
 SALARY DECIMAL(5,2),
 DEPTNO SMALLINT)

CREATE TABLE EMP_ACT
(ACNT_NUM SMALLINT GENERATED ALWAYS AS IDENTITY (START WITH 1),
 EMPNO SMALLINT)

CREATE TABLE ACCT_LOG
(ID SMALLINT GENERATED ALWAYS AS IDENTITY (START WITH 100),
 ACNT_NUM SMALLINT,
 EMPNO SMALLINT)

CREATE TRIGGER NEW_HIRE
AFTER INSERT ON EMPLOYEE
REFERENCING NEW AS NEW_EMP
FOR EACH ROW
BEGIN ATOMIC

IDENTITY_VAL_LOCAL

Chapter 3. Functions   437
IDENTITY_VAL_LOCAL

```sql
INSERT INTO EMP_ACT (EMPNO) VALUES (NEW_EMP.EMPNO);
INSERT INTO ACCT_LOG (ACNT_NUM, EMPNO)
VALUES (IDENTITY_VAL_LOCAL(), NEW_EMP.EMPNO);
END
```

The first triggered insert operation inserts a row into the EMP_ACT table. The statement uses a trigger transition variable for the EMPNO column of the EMPLOYEE table to indicate that the identity value for the EMPNO column of the EMPLOYEE table is to be copied to the EMPNO column of the EMP_ACT table. The IDENTITY_VAL_LOCAL function could not be used to obtain the value assigned to the EMPNO column of the EMPLOYEE table, because an INSERT statement has not been issued at this level of the nesting. If the IDENTITY_VAL_LOCAL function were invoked in the VALUES clause of the INSERT statement for the EMP_ACT table, it would return a null value. The insert operation against the EMP_ACT table also results in the generation of a new identity value for the ACNT_NUM column.

The second triggered insert operation inserts a row into the ACCT_LOG table. The statement invokes the IDENTITY_VAL_LOCAL function to indicate that the identity value assigned to the ACNT_NUM column of the EMP_ACT table in the previous insert operation in the triggered action is to be copied to the ACNT_NUM column of the ACCT_LOG table. The EMPNO column is assigned the same value as the EMPNO column of the EMPLOYEE table.

After the following INSERT statement and all of the triggered actions have been processed:

```sql
INSERT INTO EMPLOYEE (NAME, SALARY, DEPTNO)
VALUES ('Rupert', 989.99, 50)
```

the contents of the three tables are as follows:

```sql
SELECT EMPNO, SUBSTR(NAME,1,10) AS NAME, SALARY, DEPTNO
FROM EMPLOYEE

EMPNO  NAME      SALARY  DEPTNO
------  ---------  -------  -----
1000   Rupert    989.99  50

SELECT ACNT_NUM, EMPNO
FROM EMP_ACT

ACNT_NUM EMPNO
-------- -----
1       1000

SELECT * FROM ACCT_LOG

ID  ACNT_NUM EMPNO
------  --------  -----
100   1        1000
```

The result of the IDENTITY_VAL_LOCAL function is the most recently assigned value for an identity column at the same nesting level. After processing the original INSERT statement and all of the triggered actions, the IDENTITY_VAL_LOCAL function returns a value of 1000, because this is the value that was assigned to the EMPNO column of the EMPLOYEE table.
The schema is SYSIBM.

The INITCAP function returns a string with the first character of each *word* converted to uppercase, using the UPPER function semantics, and the other characters converted to lowercase, using the LOWER function semantics. A *word* is delimited by any of the following characters:

*Table 46. Word delimiter characters*

<table>
<thead>
<tr>
<th>Character or range of characters</th>
<th>Unicode code points or range of Unicode code points</th>
</tr>
</thead>
<tbody>
<tr>
<td>(blank)</td>
<td>U+0020</td>
</tr>
<tr>
<td>! * # $ &amp; ' () + , - . /</td>
<td>U+0021 to U+002F</td>
</tr>
<tr>
<td>: ; &lt; = &gt; ? @</td>
<td>U+003A to U+0040</td>
</tr>
<tr>
<td>[ \ ] ^ _ `</td>
<td>U+005B to U+0060</td>
</tr>
<tr>
<td>{</td>
<td>] ~</td>
</tr>
<tr>
<td>Control characters, including the following SQL control characters:</td>
<td></td>
</tr>
<tr>
<td>• tab</td>
<td></td>
</tr>
<tr>
<td>• new line</td>
<td></td>
</tr>
<tr>
<td>• form feed</td>
<td></td>
</tr>
<tr>
<td>• carriage return</td>
<td></td>
</tr>
<tr>
<td>• line feed</td>
<td></td>
</tr>
</tbody>
</table>

**Note:** A character listed in the table above might not have an allocated code point in a particular database code page.

*string-expression*

An expression that returns a value that is a CHAR or VARCHAR data type. In a Unicode database, if a supplied argument is a GRAPHIC or VARGRAPHIC data type, it is first converted to VARCHAR before evaluating the function.

The data type of the result depends on the data type of *string-expression*, as described in the following table:

*Table 47. Data type of string-expression compared to the data type of the result*

<table>
<thead>
<tr>
<th>Data type of <em>string-expression</em></th>
<th>Data type of the result</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR or VARCHAR</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>GRAPHIC or VARGRAPHIC</td>
<td>VARGRAPHIC</td>
</tr>
</tbody>
</table>

The length attribute of the result is the same as the length attribute of *string-expression*.

If the argument can be null, the result can be null; if the argument is null, the result is the null value.

**Examples:**
• Input the string “a prospective book title” to return the string “A Prospective Book Title”.
  VALUES INITCAP ('a prospective book title')
  1
  ------------------------
  A Prospective Book Title
• Input the string “YOUR NAME” to return the string “Your Name”.
  VALUES INITCAP ('YOUR NAME')
  1
  ---------
  Your Name
• Input the string “my_résumé” to return the string “My_Résumé”.
  VALUES INITCAP ('my_résumé')
  1
  ---------
  My_Résumé
• Input the string “élégant” to return the string “Élégant”.
  VALUES INITCAP ('FORMAT:élégant')
  1
  ------------
  Format:Élégant
The schema is SYSIBM. The SYSFUN version of the INSERT function continues to be available.

The INSERT function returns a string where, beginning at start in source-string, length bytes have been deleted and insert-string has been inserted.

The INSERT function is identical to the OVERLAY function, except that the length argument is mandatory.

source-string
   An expression that specifies the source string. The expression must return a value that is a built-in string, numeric, or datetime data type. If the value is not a string data type, it is implicitly cast to VARCHAR before evaluating the function.

start
   An expression that returns an integer value. The integer value specifies the starting point within the source string where the deletion of bytes and the insertion of another string is to begin. The expression must return a value that is a built-in numeric, CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type. If the value is not of type INTEGER, it is implicitly cast to INTEGER before evaluating the function. The integer value must be between 1 and the length of source-string plus one (SQLSTATE 42815). If OCTETS is specified and the result is graphic data, the value must be an odd number between 1 and twice the length attribute of source-string plus one (SQLSTATE 428GC).

length
   An expression that specifies the number of code units (in the specified string units) that are to be deleted from the source string, starting at the position identified by start. The expression must return a value that is a built-in numeric, CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type. If the value is not of type INTEGER, it is implicitly cast to INTEGER before evaluating the function. The value of the integer must be in the range of 0 to the length of source-string, expressed in units that are either implicitly or explicitly specified (SQLSTATE 22011). If OCTETS is specified and the result is graphic data, the value must be an even number between 0 and twice the length attribute of source-string (SQLSTATE 428GC).

insert-string
   An expression that specifies the string to be inserted into source-string, starting at the position identified by start. The expression must return a value that is a built-in string, numeric, or datetime data type. If the value is not a string data type, it is implicitly cast to VARCHAR before evaluating the function.

CODEUNITS16, CODEUNITS32, or OCTETS
   Specifies the string unit of start and length.
CODEUNITS16 specifies that \textit{start} and \textit{length} are expressed in 16-bit UTF-16 code units. CODEUNITS32 specifies that \textit{start} and \textit{length} are expressed in 32-bit UTF-32 code units. OCTETS specifies that \textit{start} and \textit{length} are expressed in bytes.

If the string unit is specified as CODEUNITS16 or CODEUNITS32, and the result is a binary string or bit data, an error is returned (SQLSTATE 428GC). If the string unit is specified as OCTETS, and \textit{insert-string} and \textit{source-string} are binary strings, an error is returned (SQLSTATE 42815). If the string unit is specified as OCTETS, the operation is performed in the code page of the \textit{source-string}. If a string unit is not explicitly specified, the data type of the result determines the unit that is used. If the result is graphic data, \textit{start} and \textit{length} are expressed in two-byte units; otherwise, they are expressed in bytes.

For more information about CODEUNITS16, CODEUNITS32, and OCTETS, see “String units in built-in functions” in “Character strings”.

The data type of the result depends on the data types of \textit{source-string} and \textit{insert-string}, as shown in the following table of supported type combinations.

\begin{center}
\begin{tabular}{|l|l|l|}
\hline
\textbf{source-string} & \textbf{insert-string} & \textbf{Result} \\
\hline
CHAR or VARCHAR & CHAR or VARCHAR & VARCHAR \\
GRAPHIC or VARGRAPHIC & GRAPHIC or VARGRAPHIC & VARGRAPHIC \\
CLOB & CHAR, VARCHAR, or CLOB & CLOB \\
DBCLOB & GRAPHIC, VARGRAPHIC, or DBCLOB & DBCLOB \\
CHAR or VARCHAR & CHAR FOR BIT DATA or VARCHAR FOR BIT DATA & VARCHAR FOR BIT DATA \\
CHAR FOR BIT DATA or VARCHAR FOR BIT DATA & CHAR, VARCHAR, CHAR FOR BIT DATA, or VARCHAR FOR BIT DATA & VARCHAR FOR BIT DATA \\
BLOB & BLOB & BLOB \\
\hline
\end{tabular}
\end{center}

For Unicode databases only:

\begin{center}
\begin{tabular}{|l|l|l|}
\hline
\textbf{source-string} & \textbf{insert-string} & \textbf{Result} \\
\hline
CHAR or VARCHAR & GRAPHIC or VARGRAPHIC & VARCHAR \\
GRAPHIC or VARGRAPHIC & CHAR or VARCHAR & VARGRAPHIC \\
CLOB & GRAPHIC, VARGRAPHIC, or DBCLOB & CLOB \\
DBCLOB & CHAR, VARCHAR, or CLOB & DBCLOB \\
\hline
\end{tabular}
\end{center}

A \textit{source-string} can have a length of 0; in this case, \textit{start} must be 1 and \textit{length} must be 0 (as implied by the bounds for \textit{start} and \textit{length} described above), and the result of the function is a copy of the \textit{insert-string}.

An \textit{insert-string} can also have a length of 0. This has the effect of deleting the code units from positions \textit{start} to \textit{start} + \textit{length} - 1 from the \textit{source-string}.

The length attribute of the result is the length attribute of \textit{source-string} plus the length attribute of \textit{insert-string}. The actual length of the result is $A1 - \text{MIN}(A1 - V2 + 1, V3) + A4$, where:

- $A1$ is the actual length of \textit{source-string}
- $V2$ is the value of \textit{start}
• V3 is the value of `length`
• A4 is the actual length of `insert-string`

If the actual length of the result string exceeds the maximum for the return data type, an error is returned (SQLSTATE 54006).

If any argument can be null, the result can be null; if any argument is null, the result is the null value.

Examples:
• Create the strings 'INSISTING', 'INSISERTING', and 'INSTING' from the string 'INSERTING' by inserting text into the middle of the existing text.
  ```sql
  SELECT INSERT('INSERTING', 4, 2, 'IS'),
         INSERT('INSERTING', 4, 0, 'IS'),
         INSERT('INSERTING', 4, 2, '')
  FROM SYSIBM.SYSDUMMY1
  ```

• Create the strings 'XXINSERTING', 'XXNSERTING', 'XXSERTING', and 'XXERTING' from the string 'INSERTING' by inserting text before the existing text, using 1 as the starting point.
  ```sql
  SELECT INSERT('INSERTING', 1, 0, 'XX'),
         INSERT('INSERTING', 1, 1, 'XX'),
         INSERT('INSERTING', 1, 2, 'XX'),
         INSERT('INSERTING', 1, 3, 'XX')
  FROM SYSIBM.SYSDUMMY1
  ```

• Create the string 'ABCABCXX' from the string 'ABCABC' by inserting text after the existing text. Because the source string is 6 characters long, set the starting position to 7 (one plus the length of the source string).
  ```sql
  SELECT INSERT('ABCABC', 7, 0, 'XX')
  FROM SYSIBM.SYSDUMMY1
  ```

• Change the string 'Hegelstraße' to 'Hegelstrasse'.
  ```sql
  SELECT INSERT('Hegelstraße', 10, 1, 'ss', CODEUNITS16)
  FROM SYSIBM.SYSDUMMY1
  ```

• The following example works with the Unicode string '&N~AB', where '&' is the musical symbol G clef character, and '~' is the combining tilde character. This string is shown below in different Unicode encoding forms:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>'&amp;'</code></td>
<td><code>'N'</code></td>
<td><code>~</code></td>
<td><code>'A'</code></td>
</tr>
<tr>
<td>UTF-8</td>
<td>X'F09D849E'</td>
<td>X'4E'</td>
<td>X'CC83'</td>
</tr>
<tr>
<td>UTF-16BE</td>
<td>X'D834DD1E'</td>
<td>X'004E'</td>
<td>X'0303'</td>
</tr>
</tbody>
</table>

Assume that the variables UTF8_VAR and UTF16_VAR contain the UTF-8 and the UTF-16BE representations of the string, respectively. Use the INSERT function to insert a 'C' into the Unicode string '&N~AB'.

```sql
SELECT INSERT(UTF8_VAR, 1, 4, 'C', CODEUNITS16),
       INSERT(UTF8_VAR, 1, 4, 'C', CODEUNITS32),
       INSERT(UTF8_VAR, 1, 4, 'C', OCTETS)
FROM SYSIBM.SYSDUMMY1
```

returns the values 'CAB', 'CB', and 'CN~AB', respectively.

```sql
SELECT INSERT(UTF8_VAR, 5, 1, 'C', CODEUNITS16),
       INSERT(UTF8_VAR, 5, 1, 'C', CODEUNITS32),
       INSERT(UTF8_VAR, 5, 1, 'C', OCTETS)
FROM SYSIBM.SYSDUMMY1
```

returns the values '&N~CB', '&N~AC', and '&C~AB', respectively.
INSERT

```
SELECT INSERT(UTF16_VAR, 1, 4, 'C', CODEUNITS16),
       INSERT(UTF16_VAR, 1, 4, 'C', CODEUNITS32),
       INSERT(UTF16_VAR, 1, 4, 'C', OCTETS)
FROM SYSIBM.SYSDUMMY1
```

returns the values 'CAB', 'CB', and 'CN~AB', respectively.

```
SELECT INSERT(UTF16_VAR, 5, 2, 'C', CODEUNITS16),
       INSERT(UTF16_VAR, 5, 1, 'C', CODEUNITS32),
       INSERT(UTF16_VAR, 5, 4, 'C', OCTETS)
FROM SYSIBM.SYSDUMMY1
```

returns the values '&N~C', '&N~AC', and '&CAB', respectively.
The schema is SYSIBM

The INSTR function returns the starting position of a string (called the search-string) within another string (called the source-string).

The INSTR scalar function is a synonym for the LOCATE_IN_STRING scalar function.
The schema is SYSIBM.

The INTEGER function returns an integer representation of:

- A number
- A string representation of a number
- A date value
- A time value

**Numeric to Integer:**

\[
\text{INTEGER}\left(\text{numeric-expression}\right)\]

An expression that returns a value of any built-in numeric data type.

The result is the same number that would occur if the argument were assigned to a large integer column or variable. The fractional part of the argument is truncated. If the whole part of the argument is not within the range of integers, an error is returned (SQLSTATE 22003).

**String to Integer:**

\[
\text{INTEGER}\left(\text{string-expression}\right)\]

An expression that returns a value that is a character-string or Unicode graphic-string representation of a number with a length not greater than the maximum length of a character constant.

The result is the same number that would result from \(\text{CAST}\left(\text{string-expression}\text{ AS INTEGER}\right)\). Leading and trailing blanks are eliminated and the resulting string must conform to the rules for forming an integer, decimal, floating-point, or decimal floating-point constant (SQLSTATE 22018). If the whole part of the
**INTEGER or INT**

argument is not within the range of integers, an error is returned (SQLSTATE 22003). The data type of string-expression must not be CLOB or DBCLOB (SQLSTATE 42884).

**Date to Integer:**

date-expression
An expression that returns a value of the DATE data type. The result is an INTEGER value representing the date as yyyyymmdd.

**Time to Integer:**

time-expression
An expression that returns a value of the TIME data type. The result is an INTEGER value representing the time as hhmmss.

The result of the function is a large integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

**Note:** The CAST specification should be used to increase the portability of applications. For more information, see “CAST specification”.

**Examples**

- Using the EMPLOYEE table, select a list containing salary (SALARY) divided by education level (EDLEVEL). Truncate any decimal in the calculation. The list should also contain the values used in the calculation and employee number (EMPNO). The list should be in descending order of the calculated value.

  ```sql
  SELECT INTEGER(SALARY / EDLEVEL), SALARY, EDLEVEL, EMPNO
  FROM EMPLOYEE
  ORDER BY 1 DESC
  ```

- Using the EMPLOYEE table, select the EMPNO column in integer form for further processing in the application.

  ```sql
  SELECT INTEGER(EMPNO) FROM EMPLOYEE
  ```

- Assume that the column BIRTHDATE (whose data type is DATE) has an internal value equivalent to '1964-07-20'.

  ```sql
  INTEGER(BIRTHDATE)
  ```

results in the value 19640720.

- Assume that the column STARTTIME (whose data type is TIME) has an internal value equivalent to '12:03:04'.

  ```sql
  INTEGER(STARTTIME)
  ```

results in the value 120304.
JULIAN_DAY

JULIAN_DAY

➤➤JULIAN_DAY((expression))➤➤

The schema is SYSFUN.

Returns an integer value representing the number of days from January 1, 4713 B.C. (the start of the Julian date calendar) to the date value specified in the argument.

The argument must be a DATE, TIMESTAMP, or a valid character string representation of a date or timestamp that is not a CLOB. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

The result of the function is INTEGER. The result can be null; if the argument is null, the result is the null value.
The schema is SYSIBM.

The LAST_DAY scalar function returns a datetime value that represents the last day of the month of the argument.

expression
An expression that specifies the starting date. The expression must return a value of one of the following built-in data types: a DATE or a TIMESTAMP.

The result of the function has the same data type as expression, unless expression is a string, in which case the result data type is DATE. The result can be null; if the value of date-expression is null, the result is the null value.

Any hours, minutes, seconds or fractional seconds information included in expression is not changed by the function.

Examples
• Set the host variable END_OF_MONTH with the last day of the current month.
  
  ```sql
  SET :END_OF_MONTH = LAST_DAY(CURRENT_DATE);
  ```

  The host variable END_OF_MONTH is set with the value representing the end of the current month. If the current day is 2000-02-10, then END_OF_MONTH is set to 2000-02-29.

• Set the host variable END_OF_MONTH with the last day of the month in EUR format for the given date.
  
  ```sql
  SET :END_OF_MONTH = CHAR(LAST_DAY('1965-07-07'), EUR);
  ```

  The host variable END_OF_MONTH is set with the value '31.07.1965'.
The schema is SYSIBM.

The LCASE function returns a string in which all the SBCS characters have been converted to lowercase characters.

LCASE is a synonym for LOWER.
LCASE (locale sensitive)

The schema is SYSIBM.

The LCASE function returns a string in which all characters have been converted to lowercase characters using the rules associated with the specified locale.

LCASE is a synonym for LOWER.
The least function returns the minimum value in a set of values.

The arguments must be compatible and each argument must be an expression that returns a value of any data type other than ARRAY, LOB, XML, a distinct type based on any of these types, or a structured type (SQLSTATE 42815). This function cannot be used as a source function when creating a user-defined function. Because this function accepts any compatible data types as arguments, it is not necessary to create additional signatures to support user-defined distinct types.

The selected argument is converted, if necessary, to the attributes of the result. The attributes of the result are determined by all the operands based on the rules for result data types.

The result of the function is the smallest argument value. If at least one argument can be null, the result can be null; if either argument is null, the result is the null value.

The LEAST scalar function is a synonym for the MIN scalar function.

Examples:

Assume that table T1 contains three columns C1, C2, and C3 with values 1, 7, and 4, respectively. The query:

```sql
SELECT LEAST (C1, C2, C3) FROM T1
```

returns 1.

If column C3 has a value of NULL instead of 4, the same query returns NULL.
The schema is SYSIBM. The SYSFUN version of the LEFT function continues to be available.

The LEFT function returns the leftmost string of `string-expression` of length `length`, expressed in the specified string unit. If `string-expression` is a character string, the result is a character string. If `string-expression` is a graphic string, the result is a graphic string.

`string-expression` is an expression that specifies the string from which the result is derived. The expression must return a value that is a built-in string, numeric, or datetime data type. If the value is not a string data type, it is implicitly cast to VARCHAR before evaluating the function. A substring of `string-expression` is zero or more contiguous code points of `string-expression`.

`length` is an expression that specifies the length of the result. The expression must return a value that is a built-in numeric, CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type. If the value is not of type INTEGER, it is implicitly cast to INTEGER before evaluating the function. The value must be between 0 and the length of `string-expression`, expressed in units that are either implicitly or explicitly specified (SQLSTATE 22011). If OCTETS is specified and the result is graphic data, the value must be an even number between 0 and twice the length attribute of `string-expression` (SQLSTATE 428GC).

**CODEUNITS16, CODEUNITS32, or OCTETS**

Specifies the string unit of `length`.

CODEUNITS16 specifies that `length` is expressed in 16-bit UTF-16 code units. CODEUNITS32 specifies that `length` is expressed in 32-bit UTF-32 code units. OCTETS specifies that `length` is expressed in bytes.

If the string unit is specified as CODEUNITS16 or CODEUNITS32, and `string-expression` is a binary string or bit data, an error is returned (SQLSTATE 428GC). If the string unit is specified as OCTETS and `string-expression` is a graphic string, `length` must be an even number; otherwise, an error is returned (SQLSTATE 428GC). If a string unit is not explicitly specified, the data type of the result determines the unit that is used. If the result is graphic data, `length` is expressed in two-byte units; otherwise, it is expressed in bytes. For more information about CODEUNITS16, CODEUNITS32, and OCTETS, see “String units in built-in functions” in “Character strings”.

The `string-expression` is padded on the right with the necessary number of padding characters so that the specified substring of `string-expression` always exists. The character used for padding is the same character that is used to pad the string in contexts where padding would occur. For more information on padding, see “String assignments” in “Assignments and comparisons”.
The result of the function is a varying-length string with a length attribute that is the same as the length attribute of `string-expression` and a data type that depends on the data type of `string-expression`:

- VARCHAR if `string-expression` is CHAR or VARCHAR
- CLOB if `string-expression` is CLOB
- VARGRAPHIC if `string-expression` is GRAPHIC or VARGRAPHIC
- DBCLOB if `string-expression` is DBCLOB
- BLOB if `string-expression` is BLOB

The actual length of the result (in string units) is `length`.

If any argument can be null, the result can be null; if any argument is null, the result is the null value.

Examples:

- Assume that variable `ALPHA` has a value of `'ABCDEF'`. The following statement:
  ```sql
  SELECT LEFT(ALPHA,3)
  FROM SYSIBM.SYSDUMMY1
  ```
  returns `'ABC'`, which are the three leftmost characters in `ALPHA`.

- Assume that variable `NAME`, which is defined as VARCHAR(50), has a value of `'KATIE AUSTIN'`, and that the integer variable `FIRSTNAME_LEN` has a value of 5. The following statement:
  ```sql
  SELECT LEFT(NAME,FIRSTNAME_LEN)
  FROM SYSIBM.SYSDUMMY1
  ```
  returns the value `'KATIE'`.

- The following statement returns a zero-length string.
  ```sql
  SELECT LEFT('ABCABC',0)
  FROM SYSIBM.SYSDUMMY1
  ```

- The `FIRSTNME` column in the `EMPLOYEE` table is defined as VARCHAR(12). Find the first name of an employee whose last name is 'BROWN' and return the first name in a 10-byte string.
  ```sql
  SELECT LEFT(FIRSTNME, 10)
  FROM EMPLOYEE
  WHERE LASTNAME = 'BROWN'
  ```
  returns a VARCHAR(12) string that has the value 'DAVID' followed by five blank characters.

- In a Unicode database, `FIRSTNAME` is a VARCHAR(12) column. One of its values is the 6-character string 'Jürgen'. When `FIRSTNAME` has this value:

<table>
<thead>
<tr>
<th>Function</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEFT(FIRSTNAME,2,CODEUNITS32)</td>
<td>'Jů' -- x'4AC3BC'</td>
</tr>
<tr>
<td>LEFT(FIRSTNAME,2,CODEUNITS16)</td>
<td>'Jů' -- x'4AC3BC'</td>
</tr>
<tr>
<td>LEFT(FIRSTNAME,2,octets)</td>
<td>'J' -- x'4A20', a truncated string</td>
</tr>
</tbody>
</table>

- The following example works with the Unicode string '&N~AB', where '&' is the musical symbol G clef character, and '~' is the combining tilde character. This string is shown below in different Unicode encoding forms:

<table>
<thead>
<tr>
<th></th>
<th>'A'</th>
<th>'B'</th>
</tr>
</thead>
<tbody>
<tr>
<td>UTF-8</td>
<td>X'F09D849E'</td>
<td>X'4E'</td>
</tr>
<tr>
<td>UTF-16BE</td>
<td>X'D834DD1E'</td>
<td>X'004E'</td>
</tr>
</tbody>
</table>
Assume that the variable UTF8_VAR, with a length attribute of 20 bytes, contains the UTF-8 representation of the string.

```
SELECT LEFT(UTF8_VAR, 2, CODEUNITS16),
       LEFT(UTF8_VAR, 2, CODEUNITS32),
       LEFT(UTF8_VAR, 2, OCTETS)
FROM SYSIBM.SYSDUMMY1
```

returns the values ‘&’, ‘&N’, and ‘bb’, respectively, where ‘b’ represents the blank character.

```
SELECT LEFT(UTF8_VAR, 5, CODEUNITS16),
       LEFT(UTF8_VAR, 5, CODEUNITS32),
       LEFT(UTF8_VAR, 5, OCTETS)
FROM SYSIBM.SYSDUMMY1
```

returns the values ‘&N~A’, ‘&N~AB’, and ‘&N’, respectively.

```
SELECT LEFT(UTF8_VAR, 10, CODEUNITS16),
       LEFT(UTF8_VAR, 10, CODEUNITS32),
       LEFT(UTF8_VAR, 10, OCTETS)
FROM SYSIBM.SYSDUMMY1
```

returns the values ‘&N~ABbbbb’, ‘&N~ABbbbbb’, and ‘&N~ABb’, respectively, where ‘b’ represents the blank character.

Assume that the variable UTF16_VAR, with a length attribute of 20 code units, contains the UTF-16BE representation of the string.

```
SELECT LEFT(UTF16_VAR, 2, CODEUNITS16),
       LEFT(UTF16_VAR, 2, CODEUNITS32),
       HEX (LEFT(UTF16_VAR, 2, OCTETS))
FROM SYSIBM.SYSDUMMY1
```

returns the values ‘&’, ‘&N’, and X’D834’, respectively, where X’D834’ is an unmatched high surrogate.

```
SELECT LEFT(UTF16_VAR, 5, CODEUNITS16),
       LEFT(UTF16_VAR, 5, CODEUNITS32),
       LEFT(UTF16_VAR, 6, OCTETS)
FROM SYSIBM.SYSDUMMY1
```

returns the values ‘&N~A’, ‘&N~AB’, and ‘&N’, respectively.

```
SELECT LEFT(UTF16_VAR, 10, CODEUNITS16),
       LEFT(UTF16_VAR, 10, CODEUNITS32),
       LEFT(UTF16_VAR, 10, OCTETS)
FROM SYSIBM.SYSDUMMY1
```

returns the values ‘&N~ABbbbb’, ‘&N~ABbbbbb’, and ‘&N~A’, respectively, where ‘b’ represents the blank character.
The schema is SYSIBM.

The LENGTH function returns the length of expression in the implicit or explicit string unit.

expression

An expression that returns a value that is a built-in data type. If expression can be null, the result can be null; if expression is null, the result is the null value.

**CODEUNITS16, CODEUNITS32, or OCTETS**

Specifies the string unit of the result. CODEUNITS16 specifies that the result is to be expressed in 16-bit UTF-16 code units. CODEUNITS32 specifies that the result is to be expressed in 32-bit UTF-32 code units. OCTETS specifies that the result is to be expressed in bytes.

If a string unit is explicitly specified, and if expression is not string data, an error is returned (SQLSTATE 428GC). If a string unit is specified as CODEUNITS16 or CODEUNITS32, and expression is a binary string or bit data, an error is returned (SQLSTATE 428GC). If a string unit is specified as OCTETS and expression is a binary string, an error is returned (SQLSTATE 42815). For more information about CODEUNITS16, CODEUNITS32, and OCTETS, see “String units in built-in functions” in “Character strings”.

If a string unit is not explicitly specified, the data type of the result determines the unit that is used. If the result is graphic data, the value returned specifies the length in 2-byte units. Otherwise, the value returned specifies the length in bytes.

The result of the function is a large integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The length of character and graphic strings includes trailing blanks. The length of binary strings includes binary zeroes. The length of varying-length strings is the actual length and not the maximum length. The length of all other values is the number of bytes used to represent the value:

- 2 for small integer
- 4 for large integer
- \((p/2)+1\) for decimal numbers with precision \(p\)
- 8 for DECFLOAT(16)
- 16 for DECFLOAT(34)
- The length of the string for binary strings
- The length of the string for character strings
- 4 for single-precision floating-point
- 8 for double-precision floating-point
- 4 for DATE
- 3 for TIME
- \(7+(p+1)/2\) for TIMESTAMPP\(p\)
Examples
- Assume that the host variable ADDRESS is a varying-length character string with a value of '895 Don Mills Road'.
  \[ \text{LENGTH}(\text{:ADDRESS}) \]
  returns the value 18.
- Assume that START_DATE is a column of type DATE.
  \[ \text{LENGTH}(\text{START_DATE}) \]
  returns the value 4.
- \[ \text{LENGTH}(\text{CHAR}(\text{START_DATE}, \text{EUR})) \]
  returns the value 10.
- The following examples work with the Unicode string '&N~AB', where '&' is the musical symbol G clef character, and '~' is the combining tilde character. This string is shown below in different Unicode encoding forms:

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>'&amp;'</td>
<td>'N'</td>
<td>'~'</td>
<td>'A'</td>
<td>'B'</td>
</tr>
<tr>
<td>UTF-8</td>
<td>X'F09D849E'</td>
<td>X'4E'</td>
<td>X'CC83'</td>
<td>X'41'</td>
</tr>
<tr>
<td>UTF-16BE</td>
<td>X'D834DD1E'</td>
<td>X'004E'</td>
<td>X'0303'</td>
<td>X'0041'</td>
</tr>
<tr>
<td>UTF-32BE</td>
<td>X'0001D11E'</td>
<td>X'0000004E'</td>
<td>X'00000303'</td>
<td>X'00000041'</td>
</tr>
</tbody>
</table>

Assume that the variable UTF8_VAR contains the UTF-8 representation of the string.

\[
\text{SELECT LENGTH(UFT8\_VAR, CODEUNITS16),}
\text{LENGTH(UFT8\_VAR, CODEUNITS32),}
\text{LENGTH(UFT8\_VAR, OCTETS)}
\]
\[ \text{FROM SYSIMB.SYSDUMMY1} \]

returns the values 6, 5, and 9, respectively.

Assume that the variable UTF16\_VAR contains the UTF-16BE representation of the string.

\[
\text{SELECT LENGTH(UFT16\_VAR, CODEUNITS16),}
\text{LENGTH(UFT16\_VAR, CODEUNITS32),}
\text{LENGTH(UFT16\_VAR, OCTETS)}
\]
\[ \text{FROM SYSIMB.SYSDUMMY1} \]

returns the values 6, 5, and 12, respectively.
The schema is SYSIBM. (The SYSFUN version of the LN function continues to be available.)

The LN function returns the natural logarithm of a number. The LN and EXP functions are inverse operations.

The argument must be an expression that returns a value of any built-in numeric data type. If the argument is decimal floating-point, the operation is performed in decimal floating-point; otherwise, the argument is converted to double-precision floating-point for processing by the function. The value of the argument must be greater than zero (SQLSTATE 22003).

If the argument is DECFLOAT(n), the result is DECFLOAT(n); otherwise, the result is a double-precision floating-point number. The result can be null; if the argument is null, the result is the null value.

Notes

- **Results involving DECFLOAT special values:** For decimal floating-point values, the special values are treated as follows:
  - LN(NaN) returns NaN.
  - LN(-NaN) returns -NaN.
  - LN(Infinity) returns Infinity.
  - LN(-Infinity) returns NaN and a warning.
  - LN(sNaN) returns NaN and a warning.
  - LN(-sNaN) returns -NaN and a warning.
  - LN(DECFLOAT('0') returns -Infinity.

- **Syntax alternatives:** LOG can be specified in place of LN. It is supported only for compatibility with previous versions of DB2 products. LN should be used instead of LOG, because some database managers and applications implement LOG as the common logarithm of a number instead of the natural logarithm of a number.

Example:

- Assume that NATLOG is a DECIMAL(4,2) host variable with a value of 31.62.

  ```sql
  VALUES LN(:NATLOG)
  ```

  Returns the approximate value 3.45.
LOCATE

The schema is SYSIBM. The SYSFUN version of the LOCATE function continues to be available, but it is not sensitive to the database collation.

The LOCATE function returns the starting position of the first occurrence of one string (called the search-string) within another string (called the source-string). If the search-string is not found and neither argument is null, the result is zero. If the search-string is found, the result is a number from 1 to the actual length of the source-string. The search is done using the collation of the database, unless search-string or source-string is defined as FOR BIT DATA, in which case the search is done using a binary comparison.

If the optional start is specified, it indicates the character position in the source-string at which the search is to begin. An optional string unit can be specified to indicate in what units the start and result of the function are expressed.

If the search-string has a length of zero, the result returned by the function is 1. Otherwise, if the source-string has a length of zero, the result returned by the function is 0. Otherwise:

- If the value of search-string is equal to an identical length of substring of contiguous positions within the value of source-string, the result returned by the function is the starting position of the first such substring within the source-string value.
- Otherwise, the result returned by the function is 0.

search-string
An expression that specifies the string that is the object of the search. The expression must return a value that is a built-in CHAR, VARCHAR, GRAPHIC, VARGRAPHIC, BLOB, numeric, or datetime data type. If the value is not a CHAR, VARCHAR, GRAPHIC, VARGRAPHIC, or BLOB data type, it is implicitly cast to VARCHAR before evaluating the function. The expression cannot be a BLOB file reference variable. The expression can be specified by any of the following:

- A constant
- A special register
- A global variable
- A host variable
- A scalar function whose operands are any of the above
- An expression that concatenates (using CONCAT or ||) any of the above
- An SQL procedure parameter

These rules are similar to those that are described for pattern-expression for the LIKE predicate.
LOCATE

source-string
An expression that specifies the string in which the search is to take place. The
expression must return a value that is a built-in string, numeric, or datetime
data type. If the value is not a string data type, it is implicitly cast to
VARCHAR before evaluating the function. The expression can be specified by
any of the following:

- A constant
- A special register
- A global variable
- A host variable (including a locator variable or a file reference variable)
- A scalar function
- A large object locator
- A column name
- An expression that concatenates (using CONCAT or ||) any of the above

start
An expression that specifies the position within source-string at which the
search is to start. The expression must return a value that is a built-in numeric,
CHAR, VARCHAR, GRAPHIC or VARGRAPHIC data type. If the value is not
of type INTEGER, it is implicitly cast to INTEGER before evaluating the
function. The value of the integer must be greater than or equal to zero. If start
is specified, the LOCATE function is similar to:

\[
\text{POSITION}(\text{search-string}, \text{SUBSTRING}(\text{source-string}, \text{start}, \text{string-unit}), \text{string-unit}) + \text{start} - 1
\]

where string-unit is either CODEUNITS16, CODEUNITS32, or OCTETS.

If start is not specified, the search begins at the first position of the source
string, and the LOCATE function is similar to:

\[
\text{POSITION}(\text{search-string}, \text{source-string}, \text{string-unit})
\]

CODEUNITS16, CODEUNITS32, or OCTETS

Specifies the string unit of start and the result. CODEUNITS16 specifies that
start and the result are to be expressed in 16-bit UTF-16 code units.
CODEUNITS32 specifies that start and the result are to be expressed in 32-bit
UTF-32 code units. OCTETS specifies that start and the result are to be
expressed in bytes.

If a string unit is specified as CODEUNITS16 or CODEUNITS32, and
search-string or source-string is a binary string or bit data, an error is returned
(SQLSTATE 428GC). If a string unit is specified as OCTETS and search-string
and source-string are binary strings, an error is returned (SQLSTATE 42815).

If a string unit is not explicitly specified, the data type of the result determines
the unit that is used. If the result is graphic data, start and the returned
position are expressed in two-byte units; otherwise, they are expressed in
bytes.

If a locale-sensitive UCA-based collation is used for this function, then the
CODEUNITS16 option offers the best performance characteristics.

For more information about CODEUNITS16, CODEUNITS32, and OCTETS, see
“String units in built-in functions” in “Character strings”.

The first and second arguments must have compatible string types. For more
information on compatibility, see “Rules for string conversions”. In a Unicode
database, if one string argument is character (not FOR BIT DATA) and the other
string argument is graphic, then the search-string is converted to the data type of
the source-string for processing. If one argument is character FOR BIT DATA, the
other argument must not be graphic (SQLSTATE 42846).

The result of the function is a large integer. If any argument can be null, the result
can be null; if any argument is null, the result is the null value.

Examples:
• Find the location of the first occurrence of the character ‘N’ in the string
  ‘DINING’.

  SELECT LOCATE('N', 'DINING')
  FROM SYSIBM.SYSDUMMY1

  The result is the value 3.

• For all the rows in the table named IN_TRAY, select the RECEIVED column, the
SUBJECT column, and the starting position of the string ‘GOOD’ within the
NOTE_TEXT column.

  SELECT RECEIVED, SUBJECT, LOCATE('GOOD', NOTE_TEXT)
  FROM IN_TRAY
  WHERE LOCATE('GOOD', NOTE_TEXT) <> 0

• Locate the character ‘ß’ in the string ‘Jürgen lives on Hegelstraße’, and set the
host variable LOCATION with the position, as measured in CODEUNITS32
units, within the string.

  SET :LOCATION = LOCATE('ß', 'Jürgen lives on Hegelstraße', 1, CODEUNITS32)

  The value of host variable LOCATION is set to 26.

• Locate the character ‘ß’ in the string ‘Jürgen lives on Hegelstraße’, and set the
host variable LOCATION with the position, as measured in CODEUNITS16
units, within the string.

  SET :LOCATION = LOCATE('ß', 'Jürgen lives on Hegelstraße', 1, CODEUNITS16)

  The value of host variable LOCATION is set to 26.

• Locate the character ‘ß’ in the string ‘Jürgen lives on Hegelstraße’, and set the
host variable LOCATION with the position, as measured in OCTETS, within the
string.

  SET :LOCATION = LOCATE('ß', 'Jürgen lives on Hegelstraße', 1, OCTETS)

  The value of host variable LOCATION is set to 27.

• The following examples work with the Unicode string ‘&N~AB’, where ‘&’ is
the musical symbol G clef character, and ‘~’ is the non-spacing combining tilde
character. This string is shown below in different Unicode encoding forms:

<table>
<thead>
<tr>
<th></th>
<th>‘&amp;’</th>
<th>‘N’</th>
<th>‘~’</th>
<th>‘A’</th>
<th>‘B’</th>
</tr>
</thead>
<tbody>
<tr>
<td>UTF-8</td>
<td>X’F09D849E’</td>
<td>X’4E’</td>
<td>X’CC83’</td>
<td>X’41’</td>
<td>X’42’</td>
</tr>
<tr>
<td>UTF-16BE</td>
<td>X’D834DD1E’</td>
<td>X’004E’</td>
<td>X’0303’</td>
<td>X’0041’</td>
<td>X’0042’</td>
</tr>
</tbody>
</table>

Assume that the variable UTF8_VAR contains the UTF-8 representation of the
string.

  SELECT LOCATE('~', UTF8_VAR, CODEUNITS16),
  LOCATE('~', UTF8_VAR, CODEUNITS32),
  LOCATE('~', UTF8_VAR, OCTETS)
  FROM SYSIBM.SYSDUMMY1
returns the values 4, 3, and 6, respectively.
Assume that the variable UTF16_VAR contains the UTF-16BE representation of the string.

```
SELECT LOCATE('~', UTF16_VAR, CODEUNITS16),
      LOCATE('~', UTF16_VAR, CODEUNITS32),
      LOCATE('~', UTF16_VAR, OCTETS)
FROM SYSIBM.SYSDUMMY1
```
returns the values 4, 3, and 7, respectively.

- In a Unicode database created with the case insensitive collation UCA500R1_LEN_S1, find the position of the word 'Brown' in the phrase 'The quick brown fox'.

```
SET :LOCATION = LOCATE('Brown', 'The quick brown fox', CODEUNITS16)
```

The value of the host variable LOCATION is set to 11.
LOCATE_IN_STRING

The schema is SYSIBM

The LOCATE_IN_STRING function returns the starting position of a string (called the search-string) within another string (called the source-string). If the search-string is not found and neither argument is null, the result is zero. If the search-string is found, the result is a number from 1 to the actual length of the source-string. The search is done using the collation of the database, unless search-string or source-string is defined as FOR BIT DATA, in which case the search is done using a binary comparison.

If the optional start is specified, it indicates the character position in the source-string at which the search is to begin. If the start is specified, an instance number can also be specified. The instance argument is used to determine the position of a specific occurrence of search-string within source-string. An optional string unit can be specified to indicate in what units the start and result of the function are expressed.

If the search-string has a length of zero, the result returned by the function is 1. If the source-string has a length of zero, the result returned by the function is 0. If neither condition exists, and if the value of search-string is equal to an identical length of a substring of contiguous positions within the value of source-string, the result returned by the function is the starting position of that substring within the source-string value; otherwise, the result returned by the function is 0.

source-string
An expression that specifies the string in which the search is to take place. The expression must return a value that is a built-in string, numeric, or datetime data type. If the value is not a string data type, it is implicitly cast to VARCHAR before evaluating the function. The expression can be specified in any of the following ways:

• A constant
• A special register
• A global variable
• A host variable (including a LOB locator variable or a file reference variable)
• A scalar function
• A large object locator
• A column name
• An expression that concatenates (using CONCAT or ||) any of the previous items

search-string
An expression that specifies the string that is the object of the search. The expression must return a value that is a built-in CHAR, VARCHAR, GRAPHIC, VARGRAPHIC, BLOB, numeric, or datetime data type. If the value is not a CHAR, VARCHAR, GRAPHIC, VARGRAPHIC or BLOB data type, it is
LOCATE_IN_STRING

implicitly cast to VARCHAR before evaluating the function. The actual length
must not be greater than the maximum length of a VARCHAR. The
search-string cannot be a BLOB file reference variable. The expression can be
specified in any of the following ways:

• A constant
• A special register
• A global variable
• A host variable
• A scalar function whose arguments are any of the previous items
• An expression that concatenates (using CONCAT or ||) any of the previous
  items

These rules are similar to those that are described for a pattern-expression for
the LIKE predicate.

start
An expression that specifies the position within source-string at which the
search for a match is to start. The expression must return a value that is a
built-in numeric, CHAR, VARCHAR, GRAPHIC or VARGRAPHIC data type. If
the value is not of type INTEGER, it is implicitly cast to INTEGER before
evaluating the function.

If the value of the integer is greater than zero, the search begins at start and
continues for each position to the end of the string. If the value of the integer
is less than zero, the search begins at LENGTH(source-string) + start + 1 and
continues for each position to the beginning of the string.

If start is not specified, the default is 1. If the value of the integer is zero, an
error is returned (SQLSTATE 42815).

instance
An expression that specifies which instance of search-string to search for within
source-string. The expression must return a value that is a built-in numeric,
CHAR, VARCHAR, GRAPHIC or VARGRAPHIC data type. If the value is not
of type INTEGER, it is implicitly cast to INTEGER before evaluating the
function. If instance is not specified, the default is 1. The value of the integer
must be greater than or equal to 1 (SQLSTATE 42815).

CODEUNITS16, CODEUNITS32, or OCTETS
Specifies the string unit of start and the result. CODEUNITS16 specifies that
start and the result are to be expressed in 16-bit UTF-16 code units.
CODEUNITS32 specifies that start and the result are to be expressed in 32-bit
UTF-32 code units. OCTETS specifies that start and the result are to be
expressed in bytes.

If a string unit is specified as CODEUNITS16 or CODEUNITS32, and
search-string or source-string is a binary string or bit data, an error is returned
(SQLSTATE 428GC). If a string unit is specified as OCTETS and search-string
and source-string are binary strings, an error is returned (SQLSTATE 42815).

If a string unit is not explicitly specified, the data type of the source-string
determines the string unit that is used. If the source-string is graphic data, start
and the returned position are expressed in two-byte units; otherwise, they are
expressed in bytes.

If a locale-sensitive UCA-based collation is used for this function, then the
CODEUNITS16 option offers the best performance characteristics.
LOCATE_IN_STRING

For more information about CODEUNITS16, CODEUNITS32, and OCTETS, see "String units in built-in functions" in "Character strings".

The first and second arguments must have compatible string types. For more information on compatibility, see "Rules for string conversions". In a Unicode database, if one string argument is character (not FOR BIT DATA) and the other string argument is graphic, then the search-string is converted to the data type of the source-string for processing. If one argument is character FOR BIT DATA, the other argument must not be graphic (SQLSTATE 42846).

At each search position, a match is found when the substring at that position and LENGTH(search-string) - 1 values to the right of the search position in source-string, is equal to search-string.

The result of the function is a large integer. The result is the starting position of the instance of search-string within source-string. The value is relative to the beginning of the string (regardless of the specification of start). If any argument can be null, the result can be null; if any argument is null, the result is the null value.

INSTR can be used as a synonym for LOCATE_IN_STRING.

Examples

- Locate the character 'ß' in the string 'Jürgen lives on Hegelstraße' by searching from the end of the string, and set the host variable POSITION with the position, as measured in CODEUNITS32 units, within the string.

  SET :POSITION = LOCATE_IN_STRING('Jürgen lives on Hegelstraße', 'ß', -1, CODEUNITS32);

  The value of host variable POSITION is set to 26.

- Find the location of the third occurrence of the character 'N' in the string 'WINNING' by searching from the start of the string and then set the host variable POSITION with the position of the character, as measured in bytes, within the string.

  SET :POSITION = LOCATE_IN_STRING('WINNING', 'N', 1, 3, OCTETS);

  The value of host variable POSITION is set to 6.
The schema is SYSIBM. (The SYSFUN version of the LOG10 function continues to be available.)

The LOG10 function returns the common logarithm (base 10) of a number.

The argument must be an expression that returns a value of any built-in numeric data type. If the argument is decimal floating-point, the operation is performed in decimal floating-point; otherwise, the argument is converted to double-precision floating-point for processing by the function. The value of the argument must be greater than zero (SQLSTATE 22003).

If the argument is DECFLOAT(n), the result is DECFLOAT(n); otherwise, the result is a double-precision floating-point number. The result can be null; if the argument is null, the result is the null value.

**Notes**
- **Results involving DECFLOAT special values:** For decimal floating-point values, the special values are treated as follows:
  - LOG10(NaN) returns NaN.
  - LOG10(-NaN) returns -NaN.
  - LOG10(Infinity) returns Infinity.
  - LOG10(-Infinity) returns NaN and a warning.
  - LOG10(sNaN) returns NaN and a warning.
  - LOG10(-sNaN) returns -NaN and a warning.
  - LOG10(DECFLOAT('0') returns -Infinity.

**Example**
- Assume that L is a DECIMAL(4,2) host variable with a value of 31.62.

  ```sql
  VALUES LOG10(:L)
  ```

  Returns the DOUBLE value +1.49996186559619E+000.
The LONG VARCHAR function is deprecated and might be removed in a future release. The function is compatible with earlier DB2 versions.
The `LONG_VARGRAPHIC` function is deprecated and might be removed in a future release. The function is compatible with earlier DB2 versions.
The schema is SYSIBM. (The SYSFUN version of this function continues to be available with support for CLOB arguments.)

The LOWER function returns a string in which all the SBCS characters have been converted to lowercase characters. That is, the characters A-Z will be converted to the characters a-z, and other characters will be converted to their lowercase equivalents, if they exist. For example, in code page 850, É maps to é. If the code point length of the result character is not the same as the code point length of the source character, the source character is not converted. Because not all characters are converted, LOWER(UPPER(string-expression)) does not necessarily return the same result as LOWER(string-expression).

The argument must be an expression whose value is a CHAR or VARCHAR data type.

The result of the function has the same data type and length attribute as the argument. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

Example:

Ensure that the characters in the value of column JOB in the EMPLOYEE table are returned in lowercase characters.

```
SELECT LOWER(JOB)
FROM EMPLOYEE
WHERE EMPNO = '000020';
```

The result is the value ‘manager’.
The schema is SYSIBM.

The LOWER function returns a string in which all characters have been converted to lowercase characters using the rules associated with the specified locale.

**string-expression**
An expression that returns a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC string. If **string-expression** is CHAR or VARCHAR, the expression must not be FOR BIT DATA (SQLSTATE 42815).

**locale-name**
A character constant that specifies the locale that defines the rules for conversion to lowercase characters. The value of **locale-name** is not case sensitive and must be a valid locale (SQLSTATE 42815). For information on valid locales and their naming, see “Locale names for SQL and XQuery”.

**code-units**
An integer constant that specifies the number of code units in the result. If specified, **code-units** must be an integer between 1 and 32,672 if the result is character data, or between 1 and 16,336 if the result is graphic data (SQLSTATE 42815). If **code-units** is not explicitly specified, it is implicitly the length attribute of **string-expression**. If OCTETS is specified and the result is graphic data, the value of **code-units** must be even (SQLSTATE 428GC).

**CODEUNITS16, CODEUNITS32, or OCTETS**
Specifies the string unit of **code-units**.

CODEUNITS16 specifies that **code-units** is expressed in 16-bit UTF-16 code units. CODEUNITS32 specifies that **code-units** is expressed in 32-bit UTF-32 code units. OCTETS specifies that **code-units** is expressed in bytes.

If a string unit is not explicitly specified, the data type of the result determines the unit that is used. If the result is graphic data, **code-units** is expressed in two-byte units; otherwise, it is expressed in bytes. For more information about CODEUNITS16, CODEUNITS32, and OCTETS, see “String units in built-in functions” in “Character strings”.

The result of the function is VARCHAR if **string-expression** is CHAR or VARCHAR, and VARGRAPHIC if **string-expression** is GRAPHIC or VARGRAPHIC.

The length attribute of the result is determined by the implicit or explicit value of **code-units**, the implicit or explicit string unit, and the result data type, as shown in the following table:

<table>
<thead>
<tr>
<th>String unit</th>
<th>Character result type</th>
<th>Graphic result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CODEUNITS16</td>
<td>MIN(<strong>code-units</strong> * 3, 32672)</td>
<td><strong>code-units</strong></td>
</tr>
<tr>
<td>CODEUNITS32</td>
<td>MIN(<strong>code-units</strong> * 4, 32672)</td>
<td>MIN(<strong>code-units</strong> * 2, 16336)</td>
</tr>
<tr>
<td>OCTETS</td>
<td><strong>code-units</strong></td>
<td>MIN(<strong>code-units</strong> / 2, 16336)</td>
</tr>
</tbody>
</table>
LOWER (locale sensitive)

The actual length of the result might be greater than the length of string-expression. If the actual length of the result is greater than the length attribute of the result, an error is returned (SQLSTATE 42815). If the number of code units in the result exceeds the value of code-units, an error is returned (SQLSTATE 42815).

If string-expression is not in UTF-16, this function performs code page conversion of string-expression to UTF-16, and of the result from UTF-16 to the code page of string-expression. If either code page conversion results in at least one substitution character, the result includes the substitution character, a warning is returned (SQLSTATE 01517), and the warning flag SQLWARN8 in the SQLCA is set to 'W'.

If the first argument can be null, the result can be null; if the first argument is null, the result is the null value.

Examples:

• Ensure that the characters in the value of column JOB in the EMPLOYEE table are returned in lowercase characters.

   ```sql
   SELECT LOWER(JOB, 'en_US')
   FROM EMPLOYEE
   WHERE EMPNO = '000020'
   ```

   The result is the value ‘manager’.

• Find the lowercase characters for all the ‘I’ characters in a Turkish string.

   ```sql
   VALUES LOWER('I/SM79000ıı', 'tr_TR', CODEUNITS16)
   ```

   The result is the string ‘ıııı’.
The LPAD function returns a string composed of `string-expression` that is padded on the left, with `pad` or blanks. The LPAD function treats leading or trailing blanks in `string-expression` as significant. Padding will only occur if the actual length of `string-expression` is less than `integer`, and `pad` is not an empty string.

**string-expression**
An expression that specifies the source string. The expression must return a value that is a built-in CHAR, VARCHAR, GRAPHIC, VARGRAPHIC, numeric, or datetime data type. If the value is not a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type, it is implicitly cast to VARCHAR before evaluating the function.

**integer**
An integer expression that specifies the length of the result. The expression must return a value that is a built-in numeric, CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type. If the value is not of type INTEGER, it is implicitly cast to INTEGER before evaluating the function. The value must be zero or a positive integer that is less than or equal to `n`, where `n` is 32 672 if `string-expression` is a character string, or 16 336 if `string-expression` is a graphic string.

**pad**
An expression that specifies the string with which to pad. The expression must return a value that is a built-in CHAR, VARCHAR, GRAPHIC, VARGRAPHIC, numeric, or datetime data type. If the value is not a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type, it is implicitly cast to VARCHAR before evaluating the function.

If `pad` is not specified, the pad character is determined as follows:
- SBCS blank character if `string-expression` is a character string.
- Ideographic blank character, if `string-expression` is a graphic string. For graphic string in an EUC database, X'3000' is used. For graphic string in a Unicode database, X'0020' is used.

The result of the function is a varying length string that has the same code page as `string-expression`. The value for `string-expression` and the value for `pad` must have compatible data types. If the `string-expression` and `pad` have different code pages, then `pad` is converted to the code page of `string-expression`. If either `string-expression` or `pad` is FOR BIT DATA, no character conversion occurs.

The length attribute of the result depends on whether the value for `integer` is available when the SQL statement containing the function invocation is compiled (for example, if it is specified as a constant or a constant expression) or available only when the function is executed (for example, if it is specified as the result of invoking a function). When the value is available when the SQL statement containing the function invocation is compiled, if `integer` is greater than zero, the length attribute of the result is `integer`. If `integer` is 0, the length attribute of the result is 1. When the value is available only when the function is executed, the length attribute of the result is determined according to the following table:
Table 50. Determining the result length when integer is available only when the function is executed

<table>
<thead>
<tr>
<th>Data type of string-expression</th>
<th>Result data type length</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR(n) or VARCHAR(n)</td>
<td>Minimum of n+100 and 32 672</td>
</tr>
<tr>
<td>GRAPHIC(n) or VARGRAPHIC(n)</td>
<td>Minimum of n+100 and 16 336</td>
</tr>
</tbody>
</table>

The actual length of the result is determined from integer. If integer is 0 the actual length is 0, and the result is the empty result string. If integer is less than the actual length of string-expression, the actual length is integer and the result is truncated.

If any argument can be null, the result can be null; if any argument is null, the result is the null value.

Examples:

- Assume that NAME is a VARCHAR(15) column that contains the values “Chris”, “Meg”, and “Jeff”. The following query will completely pad out a value on the left with periods:
  
  ```sql
  SELECT LPAD(NAME,15,'.') AS NAME FROM T1;
  ```

  returns:
  
  ```
  NAME
  ------------------
  ..........Chris
  ............Meg
  ...........Jeff
  ```

- Assume that NAME is a VARCHAR(15) column that contains the values “Chris”, “Meg”, and “Jeff”. The following query will only pad each value to a length of 5:
  
  ```sql
  SELECT LPAD(NAME,5,'.') AS NAME FROM T1;
  ```

  returns:
  
  ```
  NAME
  ------
  Chris
  ..Meg
  .Jeff
  ```

- Assume that NAME is a CHAR(15) column containing the values “Chris”, “Meg”, and “Jeff”. The LPAD function does not pad because NAME is a fixed length character field and is blank padded already. However, since the length of the result is 5, the columns are truncated:
  
  ```sql
  SELECT LPAD(NAME,5,'.') AS NAME FROM T1;
  ```

  returns:
  
  ```
  NAME
  ------
  Chris
  Meg
  Jeff
  ```

- Assume that NAME is a VARCHAR(15) column containing the values “Chris”, “Meg”, and “Jeff”. In some cases, a partial instance of the pad specification is returned:
  
  ```sql
  SELECT LPAD(NAME,15,'123') AS NAME FROM T1;
  ```

  returns:
LPAD

<table>
<thead>
<tr>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chris</td>
</tr>
<tr>
<td>Meg</td>
</tr>
<tr>
<td>Jeff</td>
</tr>
</tbody>
</table>

Assume that NAME is a VARCHAR(15) column containing the values “Chris”, “Meg”, and “Jeff”. Note that “Chris” is truncated, “Meg” is padded, and “Jeff” is unchanged:

```sql
SELECT LPAD(NAME, 4, '.') AS NAME FROM T1;
```

returns:

<table>
<thead>
<tr>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chi</td>
</tr>
<tr>
<td>.Meg</td>
</tr>
<tr>
<td>Jeff</td>
</tr>
</tbody>
</table>
LTRIM

The schema is SYSIBM. (The SYSFUN version of this function continues to be available with support for CLOB arguments.)

The LTRIM function removes blanks from the beginning of string-expression.

The expression must return a value that is a built-in CHAR, VARCHAR, GRAPHIC, VARGRAPHIC, numeric, or datetime data type. If the value is not a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type, it is implicitly cast to VARCHAR before evaluating the function.

- If the argument is a graphic string in a DBCS or EUC database, then the leading double byte blanks are removed.
- If the argument is a graphic string in a Unicode database, then the leading UCS-2 blanks are removed.
- Otherwise, the leading single byte blanks are removed.

The result data type of the function is:
- VARCHAR if the data type of string-expression is VARCHAR or CHAR
- VARGRAPHIC if the data type of string-expression is VARGRAPHIC or GRAPHIC

The length parameter of the returned type is the same as the length parameter of the argument data type.

The actual length of the result for character strings is the length of string-expression minus the number of bytes removed for blank characters. The actual length of the result for graphic strings is the length (in number of double byte characters) of string-expression minus the number of double byte blank characters removed. If all of the characters are removed, the result is an empty, varying-length string (length is zero).

If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Example:

Assume that host variable HELLO is defined as CHAR(9) and has a value of ‘Hello’.

VALUES LTRIM(:HELLO)

The result is ‘Hello’.
The schema is SYSIBM.

The MAX function returns the maximum value in a set of values.

The arguments must be compatible and each argument must be an expression that returns a value of any data type other than ARRAY, LOB, XML, a distinct type based on any of these types, or a structured type (SQLSTATE 42815). This function cannot be used as a source function when creating a user-defined function. Because this function accepts any compatible data types as arguments, it is not necessary to create additional signatures to support user-defined distinct types.

The selected argument is converted, if necessary, to the attributes of the result. The attributes of the result are determined by all the operands based on the rules for result data types.

The result of the function is the largest argument value. If at least one argument can be null, the result can be null; if either argument is null, the result is the null value.

The MAX scalar function is a synonym for the GREATEST scalar function.

Example:

Return the bonus for an employee, the greater of 500 and 5% of the employee’s salary.

```
SELECT EMPNO, MAX(SALARY * 0.05, 500)
FROM EMPLOYEE
```
The schema is SYSIBM.

The MAX_CARDINALITY function returns a value of type BIGINT representing the maximum number of elements that an array can contain. This is the cardinality that was specified in the CREATE TYPE statement for the ordinary array type.

array-variable
   An SQL variable, SQL parameter, or global variable of an array type, or a CAST specification of a parameter marker to an array type.

The result can be null; if the argument is null or an associative array, the result is the null value.

Example
   • Return the maximum cardinality of the RECENT_CALLS array variable of array type PHONENUMBERS:

      SET LIST_SIZE = MAX_CARDINALITY(RECENT_CALLS)

   The SQL variable LIST_SIZE is set to 50, which is the maximum cardinality that the array type PHONENUMBERS was defined with.
The schema is SYSIBM.

The MICROSECOND function returns the microsecond part of a value.

The argument must be a DATE, TIMESTAMP, timestamp duration, or a valid character string representation of a date or timestamp that is a CHAR or VARCHAR data type. If a supplied argument is a DATE, it is first converted to a TIMESTAMP(0) value, assuming a time of exactly midnight (00.00.00). In a Unicode database, if a supplied argument is a GRAPHIC or VARGRAPHIC data type, it is first converted to a character string before the function is executed.

The result of the function is a large integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The other rules depend on the data type of the argument:

- If the argument is a DATE, TIMESTAMP, or a valid string representation of a date or timestamp:
  - The integer ranges from 0 through 999 999.
  - If the precision of the timestamp exceeds 6, the value is truncated.
- If the argument is a duration:
  - The result reflects the microsecond part of the value which is an integer between -999 999 through 999 999. A nonzero result has the same sign as the argument.

Example

- Assume a table TABLEA contains two columns, TS1 and TS2, of type TIMESTAMP. Select all rows in which the microseconds portion of TS1 is not zero and the seconds portion of TS1 and TS2 are identical.

```
SELECT * FROM TABLEA
WHERE MICROSECOND(TS1) <> 0
AND
SECOND(TS1) = SECOND(TS2)
```
The schema is SYSFUN.

Returns an integer value in the range 0 to 86 400, representing the number of seconds between midnight and the time value specified in the argument.

An expression that returns a value that must be a DATE, TIME, or TIMESTAMP, or a valid string representation of a date, time, or timestamp that is not a CLOB or DBCLOB.

If expression is a DATE or a valid string representation of a date, it is first converted to a TIMESTAMP(0) value, assuming a time of exactly midnight (00.00.00).

Only Unicode databases support an argument that is a graphic string representation of a date, time, or timestamp. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

If expression is a character string, leading blanks are included and trailing blanks are removed prior to converting the value to a datetime value. For the valid formats of string representations of datetime values, see “String representations of datetime values” in “Datetime values”.

The result of the function is INTEGER. The result can be null; if the argument is null, the result is the null value.

Examples

• Find the number of seconds between midnight and 00:10:10, and midnight and 13:10:10.

  VALUES (MIDNIGHT_SECONDS('00:10:10'), MIDNIGHT_SECONDS('13:10:10'))

This example returns the following:

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>610</td>
<td>47410</td>
</tr>
</tbody>
</table>

Since a minute is 60 seconds, there are 610 seconds between midnight and the specified time. The same follows for the second example. There are 3600 seconds in an hour, and 60 seconds in a minute, resulting in 47 410 seconds between the specified time and midnight.

• Find the number of seconds between midnight and 24:00:00, and midnight and 00:00:00.

  VALUES (MIDNIGHT_SECONDS('24:00:00'), MIDNIGHT_SECONDS('00:00:00'))

This example returns the following:

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>86400</td>
<td>0</td>
</tr>
</tbody>
</table>

Note that these two values represent the same point in time, but return different MIDNIGHT_SECONDS values.
MIN

The schema is SYSIBM.

The MIN function returns the minimum value in a set of values.

The arguments must be compatible and each argument must be an expression that returns a value of any data type other than ARRAY, LOB, XML, a distinct type based on any of these types, or a structured type (SQLSTATE 42815). This function cannot be used as a source function when creating a user-defined function. Because this function accepts any compatible data types as arguments, it is not necessary to create additional signatures to support user-defined distinct types.

The selected argument is converted, if necessary, to the attributes of the result. The attributes of the result are determined by all the operands based on the rules for result data types.

The result of the function is the smallest argument value. If at least one argument can be null, the result can be null; if either argument is null, the result is the null value.

The MIN scalar function is a synonym for the LEAST scalar function.

Example:

Return the bonus for an employee, the LESSER of 5000 and 5% of the employee's salary.

```
SELECT EMPNO, MIN(SALARY * 0.05, 5000)
FROM EMPLOYEE
```
The schema is SYSIBM.

The MINUTE function returns the minute part of a value.

The argument must be a DATE, TIME, TIMESTAMP, time duration, timestamp duration, or a valid character string representation of a date, time, or timestamp that is not a CLOB. If a supplied argument is a DATE, it is first converted to a TIMESTAMP(0) value, assuming a time of exactly midnight (00.00.00). In a Unicode database, if a supplied argument is a GRAPHIC or VARGRAPHIC data type, it is first converted to a character string before the function is executed.

The result of the function is a large integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The other rules depend on the data type of the argument:
• If the argument is a DATE, TIME, TIMESTAMP, or valid string representation of a date, time or timestamp:
  – The result is the minute part of the value, which is an integer between 0 and 59.
• If the argument is a time duration or timestamp duration:
  – The result is the minute part of the value, which is an integer between -99 and 99. A nonzero result has the same sign as the argument.

Examples
• Using the CL_SCHED sample table, select all classes with a duration less than 50 minutes.

```sql
SELECT * FROM CL_SCHED
WHERE HOUR(ENDING - STARTING) = 0
AND
MINUTE(ENDING - STARTING) < 50
```
The schema is SYSFUN.

Returns the remainder of the first argument divided by the second argument. The result is negative only if first argument is negative.

The result of the function is:
• SMALLINT if both arguments are SMALLINT
• INTEGER if one argument is INTEGER and the other is INTEGER or SMALLINT
• BIGINT if one argument is BIGINT and the other argument is BIGINT, INTEGER or SMALLINT.

The result can be null; if any argument is null, the result is the null value.
The schema is SYSIBM.

The MONTH function returns the month part of a value.

The argument must be a DATE, TIMESTAMP, date duration, timestamp duration, or a valid character string representation of a date or timestamp that is not a CLOB. In a Unicode database, if a supplied argument is a graphic string (except DBCLOB), it is first converted to a character string before the function is executed.

The result of the function is a large integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The other rules depend on the data type of the argument:

- If the argument is a DATE, TIMESTAMP, or a valid string representation of a date or timestamp:
  - The result is the month part of the value, which is an integer between 1 and 12.

- If the argument is a date duration or timestamp duration:
  - The result is the month part of the value, which is an integer between -99 and 99. A nonzero result has the same sign as the argument.

Example

- Select all rows from the EMPLOYEE table for people who were born (BIRTHDATE) in DECEMBER.

  ```sql
  SELECT * FROM EMPLOYEE
  WHERE MONTH(BIRTHDATE) = 12
  ```
The schema is SYSIBM. The SYSFUN version of the `MONTHNAME` function continues to be available.

The `MONTHNAME` function returns a character string containing the name of the month (for example, January) for the month portion of `expression`, based on `locale-name` or the value of the special register CURRENT LOCALE LC_TIME.

`expression`

An expression that returns a value of one of the following built-in data types: a DATE, TIMESTAMP, or a valid character string representation of a date or timestamp that is not a CLOB. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

`locale-name`

A character constant that specifies the locale used to determine the language of the result. The value of `locale-name` is not case-sensitive and must be a valid locale (SQLSTATE 42815). For information on valid locales and their naming, see "Locale names for SQL and XQuery". If `locale-name` is not specified, the value of the special register CURRENT LOCALE LC_TIME is used.

The result is a varying-length character string. The length attribute is 100. If the resulting string exceeds the length attribute of the result, the result will be truncated. If the `expression` argument can be null, the result can be null; if the `expression` argument is null, the result is the null value. The code page of the result is the code page of the section.

Notes

- **Julian and Gregorian calendar**: The transition from the Julian calendar to the Gregorian calendar on 15 October 1582 is taken into account by this function. However, the SYSFUN version of the `MONTHNAME` function assumes the Gregorian calendar for all calculations.

- **Determinism**: `MONTHNAME` is a deterministic function. However, when `locale-name` is not explicitly specified, the invocation of the function depends on the value of the special register CURRENT LOCALE LC_TIME. This invocation that depends on the value of a special register cannot be used wherever special registers cannot be used (SQLSTATE 42621 or 428EC).

Example

- Assume that the variable `TMSTAMP` is defined as TIMESTAMP and has the following value: 2007–03–09–14.07.38.123456. The following examples show several invocations of the function and the resulting string values. The result type in each case is VARCHAR(100).

<table>
<thead>
<tr>
<th>Function invocation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>MONTHNAME(TMSTAMP, 'CLDR 1.5:en_US')</code></td>
<td>March</td>
</tr>
<tr>
<td><code>MONTHNAME(TMSTAMP, 'CLDR 1.5:de_DE')</code></td>
<td>Marz</td>
</tr>
<tr>
<td><code>MONTHNAME(TMSTAMP, 'CLDR 1.5:fr_FR')</code></td>
<td>mars</td>
</tr>
</tbody>
</table>
MONTHS_BETWEEN

```
MONTHS_BETWEEN(expression1, expression2)
```

The schema is SYSIBM.

The MONTHS_BETWEEN function returns an estimate of the number of months between `expression1` and `expression2`.

**expression1 or expression2**

Expressions that return a value of either a DATE or TIMESTAMP data type.

If `expression1` represents a date that is later than `expression2`, the result is positive. If `expression1` represents a date that is earlier than `expression2`, the result is negative.

- If `expression1` and `expression2` represent dates or timestamps with the same day of the month, or both arguments represent the last day of their respective months, the result is a the whole number difference based on the year and month values ignoring any time portions of timestamp arguments.
- Otherwise, the whole number part of the result is the difference based on the year and month values. The fractional part of the result is calculated from the remainder based on an assumption that every month has 31 days. If either argument represents a timestamp, the arguments are effectively processed as timestamps with maximum precision, and the time portions of these values are also considered when determining the result.

The result of the function is a DECIMAL(31,15). If either argument can be null, the result can be null. If either argument is null, the result is the null value.

**Examples**

- Calculate the number of months that project AD3100 will take. Assume that the start date is 1982-01-01 and the end date is 1983-02-01.

  ```
  SELECT MONTHS_BETWEEN (PRENDATE, PRSDATE)
  FROM PROJECT
  WHERE PROJNO='AD3100'
  
  The result is 13.000000000000000.
  
  Here are some additional examples to consider:

  **Table 51. Additional examples using MONTHS_BETWEEN**

<table>
<thead>
<tr>
<th>Value for argument e1</th>
<th>Value for argument e2</th>
<th>Value returned by MONTHS_BETWEEN(e1,e2)</th>
<th>Value returned by ROUND(MONTHS_BETWEEN(e1,e2)*31,2)</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>2005-02-02</td>
<td>2005-01-01</td>
<td>1.032258064516129</td>
<td>32.00</td>
<td></td>
</tr>
<tr>
<td>2007-11-01-09.00.00.00000</td>
<td>2007-12-07-14.30.12.12345</td>
<td>-1.200945386592741</td>
<td>-37.23</td>
<td></td>
</tr>
<tr>
<td>2007-12-13-09.40.30.000000</td>
<td>2007-11-13-08.40.30.000000</td>
<td>1.000000000000000</td>
<td>31.00</td>
<td>See Note 1</td>
</tr>
<tr>
<td>2007-03-15</td>
<td>2007-02-20</td>
<td>0.838709677419354</td>
<td>26.00</td>
<td>See Note 2</td>
</tr>
<tr>
<td>2008-02-29</td>
<td>2008-02-28-12.00.00</td>
<td>0.016129032258064</td>
<td>0.50</td>
<td></td>
</tr>
<tr>
<td>2008-03-29</td>
<td>2008-02-29</td>
<td>1.000000000000000</td>
<td>31.00</td>
<td></td>
</tr>
<tr>
<td>2008-03-30</td>
<td>2008-02-29</td>
<td>1.032258064516129</td>
<td>32.00</td>
<td></td>
</tr>
</tbody>
</table>
### MONTHS_BETWEEN

**Table 51. Additional examples using MONTHS_BETWEEN (continued)**

<table>
<thead>
<tr>
<th>Value for argument e1</th>
<th>Value for argument e2</th>
<th>Value returned by MONTHS_BETWEEN (e1,e2)</th>
<th>Value returned by ROUND (MONTHS_BETWEEN (e1,e2)*31.2)</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>2008-03-31</td>
<td>2008-02-29</td>
<td>1.000000000000000</td>
<td>31.00</td>
<td>See Note 3</td>
</tr>
</tbody>
</table>

**Note:**
1. The time difference is ignored because the day of the month is the same for both values.
2. The result is not 23 because, even though February has 28 days, the assumption is that all months have 31 days.
3. The result is not 33 because both dates are the last day of their respective month, and so the result is only based on the year and month portions.
The schema is SYSIBM.

The MULTIPLY_ALT scalar function returns the product of the two arguments. It is provided as an alternative to the multiplication operator, especially when the sum of the decimal precisions of the arguments exceeds 31.

The arguments can be any built-in numeric data type.

The result of the function is DECIMAL when both arguments are exact numeric data types (DECIMAL, BIGINT, INTEGER, or SMALLINT); otherwise the operation is carried out using decimal floating-point arithmetic and the result of the function is decimal floating-point with a precision determined by the data type of the arguments in the same way the precision is determined for decimal floating-point arithmetic. A floating-point or string argument is cast to DECFLOAT(34) before evaluating the function.

When the result of the function is DECIMAL, the precision and scale of the result are determined as follows, using the symbols $p$ and $s$ to denote the precision and scale of the first argument, and the symbols $p'$ and $s'$ to denote the precision and scale of the second argument.

- The precision is $\min(31, p + p')$
- The scale is:
  - 0 if the scale of both arguments is 0
  - $\min(31, s + s')$ if $p + p'$ is less than or equal to 31
  - $\max(\min(3, s + s'), 31 - (p - s + p' - s'))$ if $p + p'$ is greater than 31.

The result can be null if at least one argument can be null, or if the database is configured with `dft_sqlmathwarn` set to YES; the result is the null value if one of the arguments is null.

The MULTIPLY_ALT function is a preferable choice to the multiplication operator when performing decimal arithmetic where a scale of at least 3 is required and the sum of the precisions exceeds 31. In these cases, the internal computation is performed so that overflows are avoided. The final result is then assigned to the result data type, using truncation where necessary to match the scale. Note that overflow of the final result is still possible when the scale is 3.

The following is a sample comparing the result types using MULTIPLY_ALT and the multiplication operator.

<table>
<thead>
<tr>
<th>Type of argument 1</th>
<th>Type of argument 2</th>
<th>Result using MULTIPLY_ALT</th>
<th>Result using multiplication operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>DECIMAL(31,3)</td>
<td>DECIMAL(15,8)</td>
<td>DECIMAL(31,3)</td>
<td>DECIMAL(31,11)</td>
</tr>
<tr>
<td>DECIMAL(26,23)</td>
<td>DECIMAL(10,1)</td>
<td>DECIMAL(31,19)</td>
<td>DECIMAL(31,24)</td>
</tr>
<tr>
<td>DECIMAL(18,17)</td>
<td>DECIMAL(20,19)</td>
<td>DECIMAL(31,29)</td>
<td>DECIMAL(31,31)</td>
</tr>
<tr>
<td>DECIMAL(16,3)</td>
<td>DECIMAL(17,8)</td>
<td>DECIMAL(31,9)</td>
<td>DECIMAL(31,11)</td>
</tr>
<tr>
<td>DECIMAL(26,5)</td>
<td>DECIMAL(11,0)</td>
<td>DECIMAL(31,3)</td>
<td>DECIMAL(31,5)</td>
</tr>
</tbody>
</table>
**MULTIPLY_ALT**

<table>
<thead>
<tr>
<th>Type of argument 1</th>
<th>Type of argument 2</th>
<th>Result using MULTIPLY_ALT</th>
<th>Result using multiplication operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>DECIMAL(21,1)</td>
<td>DECIMAL(15,1)</td>
<td>DECIMAL(31,2)</td>
<td>DECIMAL(31,2)</td>
</tr>
</tbody>
</table>

Example:

Multiply two values where the data type of the first argument is DECIMAL(26,3) and the data type of the second argument is DECIMAL(9,8). The data type of the result is DECIMAL(31,7).

```
MULTIPLY_ALT(98765432109876543210987.654,5.43210987)
```

1

```
5365046785785294857887.5277415
```

Note that the complete product of these two numbers is 5365046785785294857887.52774154498, but the last 4 digits are truncated to match the scale of the result data type. Using the multiplication operator with the same values will cause an arithmetic overflow, since the result data type is DECIMAL(31,11) and the result value has 24 digits left of the decimal, but the result data type only supports 20 digits.
The schema is SYSIBM.

The NEXT_DAY scalar function returns a datetime value that represents the first weekday, named by string-expression, that is later than the date in expression.

expression
An expression that returns a value of one of the following built-in data types: a DATE or a TIMESTAMP.

string-expression
An expression that returns a built-in character data type. The value must be a valid day of the week for the locale-name. The value can be specified either as the full name of the day or the associated abbreviation. For example, if the locale is 'en_US' then the following values are valid:

<table>
<thead>
<tr>
<th>Day of week</th>
<th>Abbreviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>MONDAY</td>
<td>MON</td>
</tr>
<tr>
<td>TUESDAY</td>
<td>TUE</td>
</tr>
<tr>
<td>WEDNESDAY</td>
<td>WED</td>
</tr>
<tr>
<td>THURSDAY</td>
<td>THU</td>
</tr>
<tr>
<td>FRIDAY</td>
<td>FRI</td>
</tr>
<tr>
<td>SATURDAY</td>
<td>SAT</td>
</tr>
<tr>
<td>SUNDAY</td>
<td>SUN</td>
</tr>
</tbody>
</table>

The minimum length of the input value is the length of the abbreviation. The characters can be specified in lower or upper case and any characters immediately following a valid abbreviation are ignored.

locale-name
A character constant that specifies the locale used to determine the language of the string-expression value. The value of locale-name is not case sensitive and must be a valid locale (SQLSTATE 42815). For information on valid locales and their naming, see “Locale names for SQL and XQuery”. If locale-name is not specified, the value of the special register CURRENT LOCALE LC_TIME is used.

The result of the function has the same data type as expression, unless expression is a string, in which case the result data type is TIMESTAMP(6) . The result can be null; if any argument is null, the result is the null value.

Any hours, minutes, seconds or fractional seconds information included in expression is not changed by the function. If expression is a string representing a date, the time information in the resulting TIMESTAMP value is all set to zero.

Notes
• Determinism: NEXT_DAY is a deterministic function. However, when locale-name is not explicitly specified, the invocation of the function depends on
the value of the special register CURRENT LOCALE LC_TIME. Invocations of
the function that depend on the value of a special register cannot be used
wherever special registers cannot be used.

Examples

- Set the variable NEXTDAY with the date of the Tuesday following April 24,
  2007.

  \begin{verbatim}
  NEXTDAY = NEXT_DAY(DATE '2007-04-24', 'TUESDAY')
  \end{verbatim}

  The variable NEXTDAY is set with the value of '2007-05-01-00.00.00.000000',
  since April 24, 2007 is itself a Tuesday.

- Set the variable vNEXTDAY with the date of the first Monday in May, 2007.
  Assume the variable vDAYOFWEEK = 'MON'.

  \begin{verbatim}
  vNEXTDAY = NEXT_DAY(LAST_DAY(CURRENT_TIMESTAMP), vDAYOFWEEK)
  \end{verbatim}

  The variable vNEXTDAY is set with the value of '2007-05-07-12.01.01.123456',
  assuming that the value of the CURRENT_TIMESTAMP special register is
  '2007-04-24-12.01.01.123456'.


The schema is SYSIBM.

The NORMALIZE_DECFLOAT function returns a decimal floating-point value equal to the input argument set to its simplest form; that is, a nonzero number with trailing zeros in the coefficient has those zeros removed. This may require representing the number in normalized form by dividing the coefficient by the appropriate power of ten and adjusting the exponent accordingly. A zero value has its exponent set to 0.

expression
An expression that returns a value of any built-in numeric data type. Arguments of type SMALLINT, INTEGER, REAL, DOUBLE, or DECIMAL(p,s), where \( p \leq 16 \), are converted to DECFLOAT(16) for processing. Arguments of type BIGINT or DECIMAL(p,s), where \( p > 16 \), are converted to DECFLOAT(34) for processing.

The result of the function is a DECFLOAT(16) value if the data type of expression after conversion to decimal floating-point is DECFLOAT(16). Otherwise, the result of the function is a DECFLOAT(34) value. If the argument is a special decimal floating-point value, the result is the same special decimal floating-point value. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Examples:
- The following examples show the values that are returned by the NORMALIZE_DECFLOAT function, given a variety of input decimal floating-point values:

  \[
  \begin{align*}
  \text{NORMALIZE\_DEC\_FLOAT}(\text{DEC\_FLOAT}(2.1)) &= 2.1 \\
  \text{NORMALIZE\_DEC\_FLOAT}(\text{DEC\_FLOAT}(-2.0)) &= -2 \\
  \text{NORMALIZE\_DEC\_FLOAT}(\text{DEC\_FLOAT}(1.00)) &= 1.2 \\
  \text{NORMALIZE\_DEC\_FLOAT}(\text{DEC\_FLOAT}(-120)) &= -1.2E+2 \\
  \text{NORMALIZE\_DEC\_FLOAT}(\text{DEC\_FLOAT}(120.00)) &= 1.2E+2 \\
  \text{NORMALIZE\_DEC\_FLOAT}(\text{DEC\_FLOAT}(0.00)) &= 0 \\
  \text{NORMALIZE\_DEC\_FLOAT}(-\text{NAN}) &= -\text{NAN} \\
  \text{NORMALIZE\_DEC\_FLOAT}(-\text{INFINITY}) &= -\text{Infinity}
  \end{align*}
  \]
The schema is SYSIBM.

The NULLIF function returns a null value if the arguments are equal, otherwise it returns the value of the first argument.

The arguments must be comparable. They can be of either a built-in (other than a LOB string) or distinct data type (other than based on a LOB string). (This function cannot be used as a source function when creating a user-defined function. Because this function accepts any compatible data types as arguments, it is not necessary to create additional signatures to support user-defined distinct types.) The attributes of the result are the attributes of the first argument.

The result of using NULLIF(e1,e2) is the same as using the expression

```
CASE WHEN e1=e2 THEN NULL ELSE e1 END
```

Note that when e1=e2 evaluates to unknown (because one or both arguments is NULL), CASE expressions consider this not true. Therefore, in this situation, NULLIF returns the value of the first argument.

Example:

- Assume host variables PROFIT, CASH, and LOSSES have DECIMAL data types with the values 4500.00, 500.00, and 5000.00 respectively:

  ```
  NULLIF (:PROFIT + :CASH , :LOSSES )
  ```

Returns a null value.
The schema is SYSIBM.

The NVL function returns the first argument that is not null.

NVL is a synonym for COALESCE.
The schema is SYSIBM.

The OCTET_LENGTH function returns the length of \textit{expression} in octets (bytes).

\[ \text{OCTET_LENGTH(expression)} \]

\textit{expression}

An expression that returns a value that is a built-in string data type.

The result of the function is INTEGER. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The length of character or graphic strings includes trailing blanks. The length of binary strings includes binary zeroes. The length of varying-length strings is the actual length and not the maximum length.

For greater portability, code your application to be able to accept a result of data type DECIMAL(31).

Examples:

- Assume that table T1 has a GRAPHIC(10) column named C1.
  
  \[
  \text{SELECT OCTET_LENGTH}(C1) \text{ FROM T1}
  \]

  returns the value 20.

- The following example works with the Unicode string ‘\&N~AB’, where ‘\&’ is the musical symbol G clef character, and ‘~’ is the combining tilde character. This string is shown below in different Unicode encoding forms:

  \[
  \begin{array}{cccccc}
  \text{UTF-8} & X'F09D849E' & X'4E' & X'CC83' & X'41' & X'42' \\
  \text{UTF-16BE} & X'D834DD1E' & X'004E' & X'0303' & X'0041' & X'0042'
  \end{array}
  \]

  Assume that the variables UTF8_VAR and UTF16_VAR contain the UTF-8 and the UTF-16BE representations of the string, respectively.

  \[
  \text{SELECT OCTET_LENGTH(UTF8_VAR), OCTET_LENGTH(UTF16_VAR) FROM SYSCAT.SYSTABLES}
  \]

  returns the values 9 and 12, respectively.
The schema is SYSIBM.

The OVERLAY function returns a string in which, beginning at start in source-string, length of the specified code units have been deleted and insert-string has been inserted.

**source-string**

An expression that specifies the source string. The expression must return a value that is a built-in string, numeric, or datetime data type. If the value is not a string data type, it is implicitly cast to VARCHAR before evaluating the function.

**insert-string**

An expression that specifies the string to be inserted into source-string, starting at the position identified by start. The expression must return a value that is a built-in string, numeric, or datetime data type. If the value is not a string data type, it is implicitly cast to VARCHAR before evaluating the function. If the code page of the insert-string differs from that of the source-string, insert-string is converted to the code page of the source-string.

**start**

An expression that returns an integer value. The integer value specifies the starting point within the source string where the deletion of bytes and the insertion of another string is to begin. The expression must return a value that is a built-in numeric, CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type. If the value is not of type INTEGER, it is implicitly cast to INTEGER before evaluating the function. The integer value must be between 1 and the length of source-string plus one (SQLSTATE 42815). If OCTETS is specified and the result is graphic data, the value must be an odd number between 1 and twice the length attribute of source-string plus one (SQLSTATE 428GC).

**length**

An expression that specifies the number of code units (in the specified string units) that are to be deleted from the source string, starting at the position identified by start. The expression must return a value that is a built-in numeric, CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type. If the value is not of type INTEGER, it is implicitly cast to INTEGER before evaluating the function. The integer value must be between 0 and the length of source-string, expressed in units that are either implicitly or explicitly specified (SQLSTATE 22011). If OCTETS is specified and the result is graphic data, the value must be an even number between 0 and twice the length attribute of source-string (SQLSTATE 428GC).

Not specifying length is equivalent to specifying a value of 1, except when OCTETS is specified and the result is graphic data, in which case, not specifying length is equivalent to specifying a value of 2.

**CODEUNITS16, CODEUNITS32, or OCTETS**

Specifies the string unit of start and length.
OVERLAY

CODEUNITS16 specifies that \textit{start} and \textit{length} are expressed in 16-bit UTF-16 code units. CODEUNITS32 specifies that \textit{start} and \textit{length} are expressed in 32-bit UTF-32 code units. OCTETS specifies that \textit{start} and \textit{length} are expressed in bytes.

If the string unit is specified as CODEUNITS16 or CODEUNITS32, and the result is a binary string or bit data, an error is returned (SQLSTATE 428GC). If the string unit is specified as OCTETS, and \textit{insert-string} and \textit{source-string} are binary strings, an error is returned (SQLSTATE 42815). If the string unit is specified as OCTETS, the operation is performed in the code page of the \textit{source-string}. For more information about CODEUNITS16, CODEUNITS32, and OCTETS, see “String units in built-in functions” in “Character strings”.

The data type of the result depends on the data types of \textit{source-string} and \textit{insert-string}, as shown in the following table of supported type combinations.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|}
\hline
\textit{source-string} & \textit{insert-string} & Result \\
\hline
CHAR or VARCHAR & CHAR or VARCHAR & VARCHAR \\
GRAPHIC or VARGRAPHIC & GRAPHIC or VARGRAPHIC & VARGRAPHIC \\
CLOB & CHAR, VARCHAR, or CLOB & CLOB \\
DBCLOB & GRAPHIC, VARGRAPHIC, or DBCLOB & DBCLOB \\
CHAR or VARCHAR & CHAR FOR BIT DATA or VARCHAR FOR BIT DATA & VARCHAR FOR BIT DATA \\
CHAR FOR BIT DATA or VARCHAR FOR BIT DATA & CHAR, VARCHAR, CHAR FOR BIT DATA, or VARCHAR FOR BIT DATA & VARCHAR FOR BIT DATA \\
BLOB & BLOB & BLOB \\
\hline
\end{tabular}
\end{table}

\textbf{For Unicode databases only:}

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|}
\hline
\textit{source-string} & \textit{insert-string} & Result \\
\hline
CHAR or VARCHAR & GRAPHIC or VARGRAPHIC & VARCHAR \\
GRAPHIC or VARGRAPHIC & CHAR or VARCHAR & VARGRAPHIC \\
CLOB & GRAPHIC, VARGRAPHIC, or DBCLOB & CLOB \\
DBCLOB & CHAR, VARCHAR, or CLOB & DBCLOB \\
\hline
\end{tabular}
\end{table}

A \textit{source-string} can have a length of 0; in this case, \textit{start} must be 1 and \textit{length} must be 0 (as implied by the bounds for \textit{start} and \textit{length} described above), and the result of the function is a copy of the \textit{insert-string}. If length is not explicitly specified in this case, an error is returned because the assumed length is nonzero (SQLSTATE 22011).

An \textit{insert-string} can also have a length of 0. This has the effect of deleting the code units from positions \textit{start} to \textit{start} + \textit{length} - 1 from the \textit{source-string}.

The length attribute of the result is the length attribute of \textit{source-string} plus the length attribute of \textit{insert-string}. The actual length of the result is \( A1 - \text{MIN}(A1 - V2 + 1), V3) + A4 \), where:

- \( A1 \) is the actual length of \textit{source-string}
- \( V2 \) is the value of \textit{start}
• V3 is the value of \textit{length}
• A4 is the actual length of \textit{insert-string}

If the actual length of the result string exceeds the maximum for the return data type, an error is returned (SQLSTATE 54006).

If any argument can be null, the result can be null; if any argument is null, the result is the null value.

Examples:
• Create the strings ‘INSISTING’, ‘INSISERTING’, and ‘INSTING’ from the string ‘INSERTING’ by inserting text into the middle of the existing text.

\begin{verbatim}
SELECT OVERLAY('INSERTING','IS',4,2,OCTETS),
       OVERLAY('INSERTING','IS',4,0,OCTETS),
       OVERLAY('INSERTING','','',4,2,OCTETS)
FROM SYSIBM.SYSDUMMY1
\end{verbatim}

• Create the strings ‘XXINSERTING’, ‘XXNSERTING’, ‘XXSERTING’, and ‘XXERTING’ from the string ‘INSERTING’ by inserting text before the existing text, using 1 as the starting point.

\begin{verbatim}
SELECT OVERLAY('INSERTING','XX',1,0,CODEUNITS16),
       OVERLAY('INSERTING','XX',1,1,CODEUNITS16),
       OVERLAY('INSERTING','XX',1,2,CODEUNITS16),
       OVERLAY('INSERTING','XX',1,3,CODEUNITS16)
FROM SYSIBM.SYSDUMMY1
\end{verbatim}

• Create the string ‘ABCABCXX’ from the string ‘ABCABC’ by inserting text after the existing text. Because the source string is 6 characters long, set the starting position to 7 (one plus the length of the source string).

\begin{verbatim}
SELECT OVERLAY('ABCABC','XX',7,0,CODEUNITS16)
FROM SYSIBM.SYSDUMMY1
\end{verbatim}

• Change the string ‘Hegelstraße’ to ‘Hegelstrasse’.

\begin{verbatim}
SELECT OVERLAY('Hegelstraße','ss',10,1,CODEUNITS16)
FROM SYSIBM.SYSDUMMY1
\end{verbatim}

• The following example works with the Unicode string ’&N~AB’, where ’&’ is the musical symbol G clef character, and ’~’ is the combining tilde character. This string is shown below in different Unicode encoding forms:

\begin{tabular}{|c|c|c|c|c|}
\hline
  & ‘&’ & ‘N’ & ‘~’ & ‘A’ & ‘B’ \\
\hline
UTF-8 & X’F09D849E’ & X’4E’ & X’CC83’ & X’41’ & X’42’ \\
UTF-16BE & X’D834DD1E’ & X’004E’ & X’0303’ & X’0041’ & X’0042’ \\
\hline
\end{tabular}

Assume that the variables UTF8_VAR and UTF16_VAR contain the UTF-8 and the UTF-16BE representations of the string, respectively. Use the OVERLAY function to insert a ‘C’ into the Unicode string ’&N~AB’.

\begin{verbatim}
SELECT OVERLAY(UTF8_VAR, ’C’, 1, CODEUNITS16),
       OVERLAY(UTF8_VAR, ’C’, 1, CODEUNITS32),
       OVERLAY(UTF8_VAR, ’C’, 1, OCTETS)
FROM SYSIBM.SYSDUMMY1
\end{verbatim}

returns the values ‘C?N~AB’, ‘CN~AB’, and ‘CbbbN~AB’, respectively, where ‘?’ represents X’EDB49E’, which corresponds to the X’DD1E’ in the intermediate UTF-16 form, and ‘bbb’ replaces the UTF-8 incomplete characters X’9D849E’.

\begin{verbatim}
SELECT OVERLAY(UTF8_VAR, ’C’, 5, CODEUNITS16),
       OVERLAY(UTF8_VAR, ’C’, 5, CODEUNITS32),
       OVERLAY(UTF8_VAR, ’C’, 5, OCTETS)
FROM SYSIBM.SYSDUMMY1
\end{verbatim}
OVERLAY

returns the values ‘&N~CB’, ‘&N~AC’, and ‘&N~AB’, respectively.

```
SELECT OVERLAY=UTF16_VAR, 'C', 1, CODEUNITS16,
OVERLAY=UTF16_VAR, '~C', 1, CODEUNITS32
FROM SYSIBM.SYSDUMMY1
```

returns the values ‘C?N~AB’ and ‘CN~AB’, respectively, where ‘?’ represents the
unmatched low surrogate U+DD1E.

```
SELECT OVERLAY=UTF16_VAR, 'C', 5, CODEUNITS16,
OVERLAY=UTF16_VAR, '~C', 5, CODEUNITS32
FROM SYSIBM.SYSDUMMY1
```

returns the values ‘&N~CB’ and ‘&N~AC’, respectively.
PARAMETER

The PARAMETER function represents a position in an SQL statement where the value is provided dynamically by XQuery as part of the invocation of the db2-fn:sqlquery function.

The schema is SYSIBM.

The integer-constant is a position index to a value in the arguments of db2-fn:sqlquery. The value must be between 1 and the total number of the arguments specified in the db2-fn:sqlquery SQL statement (SQLSTATE 42815).

The PARAMETER function represents a position in an SQL statement where the value is provided dynamically by XQuery as part of the invocation of the db2-fn:sqlquery function. The argument of the PARAMETER function determines which value is substituted for the PARAMETER function when the db2-fn:sqlquery function is executed. The value supplied by the PARAMETER function can be referenced multiple times within the same SQL statement.

This function can only be used in a fullselect contained in the string literal argument of the db2-fn:sqlquery function in an XQuery expression (SQLSTATE 42887).

Example:

In the following example, the db2-fn:sqlquery function call uses one PARAMETER function call and the XQuery expression $po/@OrderDate, the order date attribute. The PARAMETER function call returns the value of order date attribute:

```
<query>
  declare default element namespace "http://posample.org";
  for $po in db2-fn:xmlcolumn('PURCHASEORDER.PORDER')/PurchaseOrder,
    $item in $po/item/partid
  for $p in db2-fn:sqlquery(  
    "select description from product where promostart < PARAMETER(1)",
    $po/@OrderDate )
  where $p//@pid = $item
  return
  <RESULT>
    <PoNum>{data($po/@PoNum)}</PoNum>
    <PartID>{data($item)} </PartID>
    <PoDate>{data($po/@OrderDate)}</PoDate>
  </RESULT>
</query>
```

The example returns the purchase ID, part ID, and the purchase date for all the parts sold after the promotional start date.
The schema is SYSIBM.

The POSITION function returns the starting position of the first occurrence of one string (called the search-string) within another string (called the source-string). If search-string is not found and neither argument is null, the result is zero. If the search-string is found, the result is a number from 1 to the actual length of source-string, expressed in the string unit that is explicitly specified. The search is done using the collation of the database, unless search-string or source-string is defined as FOR BIT DATA, in which case the search is done using a binary comparison.

If source-string has an actual length of 0, the result of the function is 0. If search-string has an actual length of 0 and source-string is not null, the result of the function is 1.

**search-string**

An expression that specifies the string that is the object of the search. The expression must return a value that is a built-in CHAR, VARCHAR, GRAPHIC, VARGRAPHIC, BLOB, numeric, or datetime data type. If the value is not a CHAR, VARCHAR, GRAPHIC, VARGRAPHIC or BLOB data type, it is implicitly cast to VARCHAR before evaluating the function. The expression cannot be a BLOB file reference variable. The expression can be specified by any of the following:

- A constant
- A special register
- A host variable
- A scalar function whose operands are any of the above
- An expression that concatenates (using CONCAT or ||) any of the above
- An SQL procedure parameter

These rules are similar to those that are described for pattern-expression for the LIKE predicate.

**source-string**

The expression must return a value that is a built-in string, numeric, or datetime data type. If the value is not a string data type, it is implicitly cast to VARCHAR before evaluating the function. The expression can be specified by any of the following:

- A constant
- A special register
- A host variable (including a locator variable or a file reference variable)
- A scalar function
- A large object locator
POSITION

- A column name
- An expression that concatenates (using CONCAT or ||) any of the above

**CODEUNITS16, CODEUNITS32, or OCTETS**

Specifies the string unit of the result. CODEUNITS16 specifies that the result is to be expressed in 16-bit UTF-16 code units. CODEUNITS32 specifies that the result is to be expressed in 32-bit UTF-32 code units. OCTETS specifies that the result is to be expressed in bytes.

If a string unit is specified as CODEUNITS16 or CODEUNITS32, and search-string or source-string is a binary string or bit data, an error is returned (SQLSTATE 428GC). If a string unit is specified as OCTETS and search-string and source-string are binary strings, an error is returned (SQLSTATE 42815).

If a locale-sensitive UCA-based collation is used for this function, then the CODEUNITS16 option offers the best performance characteristics.

For more information about CODEUNITS16, CODEUNITS32, and OCTETS, see “String units in built-in functions” in “Character strings”.

The first and second arguments must have compatible string types. For more information on compatibility, see “Rules for string conversions”. In a Unicode database, if one string argument is character (not FOR BIT DATA) and the other string argument is graphic, then the search-string is converted to the data type of the source-string for processing. If one argument is character FOR BIT DATA, the other argument must not be graphic (SQLSTATE 42846).

The result of the function is a large integer. If any argument can be null, the result can be null; if any argument is null, the result is the null value.

Examples:
- Select the RECEIVED column, the SUBJECT column, and the starting position of the string 'GOOD BEER' within the NOTE_TEXT column for all rows in the IN_TRAY table that contain that string.

  ```sql
  SELECT RECEIVED, SUBJECT, POSITION('GOOD BEER', NOTE_TEXT, OCTETS)
  FROM IN_TRAY
  WHERE POSITION('GOOD BEER', NOTE_TEXT, OCTETS) <> 0
  ```

- Find the position of the character 'ß' in the string 'Jürgen lives on Hegelstraße', and set the host variable LOCATION with the position, as measured in CODEUNITS32 units, within the string.

  ```sql
  SET :LOCATION = POSITION('ß', 'Jürgen lives on Hegelstraße', CODEUNITS32)
  ```

  The value of host variable LOCATION is set to 26.

- Find the position of the character 'ß' in the string 'Jürgen lives on Hegelstraße', and set the host variable LOCATION with the position, as measured in OCTETS, within the string.

  ```sql
  SET :LOCATION = POSITION('ß', 'Jürgen lives on Hegelstraße', OCTETS)
  ```

  The value of host variable LOCATION is set to 27.

- The following examples work with the Unicode string ’&N~AB’, where ‘&’ is the musical symbol G clef character, and ‘~’ is the non-spacing combining tilde character. This string is shown below in different Unicode encoding forms:

```sql
SELECT RECEIVED, SUBJECT, POSITION('GOOD BEER', NOTE_TEXT, OCTETS)
FROM IN_TRAY
WHERE POSITION('GOOD BEER', NOTE_TEXT, OCTETS) <> 0
```
Assume that the variable UTF8_VAR contains the UTF-8 representation of the string.

```
SELECT POSITION('N', UTF8_VAR, CODEUNITS16),
       POSITION('N', UTF8_VAR, CODEUNITS32),
       POSITION('N', UTF8_VAR, OCTETS)
FROM SYSIBM.SYSDUMMY1
```
returns the values 3, 2, and 5, respectively.

Assume that the variable UTF16_VAR contains the UTF-16BE representation of the string.

```
SELECT POSITION('B', UTF16_VAR, CODEUNITS16),
       POSITION('B', UTF16_VAR, CODEUNITS32),
       POSITION('B', UTF16_VAR, OCTETS)
FROM SYSIBM.SYSDUMMY1
```
returns the values 6, 5, and 11, respectively.

- In a Unicode database created with the case insensitive collation
  UCA500R1_LEN_S1, find the position of the word ‘Brown’ in the phrase ‘The quick brown fox’.

```
SET :LOCATION = POSITION('Brown', 'The quick brown fox', CODEUNITS16)
```

The value of the host variable LOCATION is set to 11.
The schema is SYSIBM.

The POSSTR function returns the starting position of the first occurrence of one string (called the search-string) within another string (called the source-string). Numbers for the search-string position start at 1 (not 0).

The result of the function is a large integer. If either of the arguments can be null, the result can be null; if either of the arguments is null, the result is the null value.

**source-string**

An expression that specifies the source string in which the search is to take place.

The expression must return a value that is a built-in string, numeric, or datetime data type. If the value is not a string data type, it is implicitly cast to VARCHAR before evaluating the function. The expression can be specified by any one of:

- A constant
- A special register
- A global variable
- A host variable (including a locator variable or a file reference variable)
- A scalar function
- A large object locator
- A column name
- An expression that concatenates (using CONCAT or ||) any of the above

**search-string**

An expression that specifies the string that is to be searched for.

The expression must return a value that is a built-in CHAR, VARCHAR, GRAPHIC, VARGRAPHIC, BLOB, numeric, or datetime data type. If the value is not a CHAR, VARCHAR, GRAPHIC, VARGRAPHIC or BLOB data type, it is implicitly cast to VARCHAR before evaluating the function. The actual length must not be greater than maximum length of a VARCHAR. The expression cannot be a BLOB file reference variable. The expression can be specified by any one of:

- A constant
- A special register
- A global variable
- A host variable
- A scalar function whose operands are any of the above
- An expression that concatenates (using CONCAT or ||) any of the above
- An SQL procedure parameter

The following are examples of invalid string expressions or strings:

- SQL user-defined function parameters
- Trigger transition variables
- Local variables in dynamic compound statements
POSSTR

In a Unicode database, if one argument is character (not FOR BIT DATA) and the other argument is graphic, then the search-string is converted to the data type of the source-string for processing. If one argument is character FOR BIT DATA, the other argument must not be graphic (SQLSTATE 42846).

Both search-string and source-string have zero or more contiguous positions. If the strings are character or binary strings, a position is a byte. If the strings are graphic strings, a position is a graphic (DBCS) character.

The POSSTR function accepts mixed data strings. However, POSSTR operates on a strict byte-count basis, oblivious to the database collation and to changes between single and multi-byte characters.

The following rules apply:
- The data types of source-string and search-string must be compatible, otherwise an error is raised (SQLSTATE 42884).
  - If source-string is a character string, then search-string must be a character string, but not a CLOB, with an actual length of 32,672 bytes or less.
  - If source-string is a graphic string, then search-string must be a graphic string, but not a DBCLOB, with an actual length of 16,336 double-byte characters or less.
  - If source-string is a binary string, then search-string must be a binary string with an actual length of 32,672 bytes or less.
- If search-string has a length of zero, the result returned by the function is 1.
- Otherwise:
  - If source-string has a length of zero, the result returned by the function is zero.
  - Otherwise:
    - If the value of search-string is equal to an identical length substring of contiguous positions from the value of source-string, then the result returned by the function is the starting position of the first such substring within the source-string value.
    - Otherwise, the result returned by the function is 0.

Example
- Select RECEIVED and SUBJECT columns as well as the starting position of the words 'GOOD BEER' within the NOTE_TEXT column for all entries in the IN_TRAY table that contain these words.

```
SELECT RECEIVED, SUBJECT, POSSTR(NOTE_TEXT, 'GOOD BEER')
FROM IN_TRAY
WHERE POSSTR(NOTE_TEXT, 'GOOD BEER') <> 0
```
```plaintext
POWER

POWER(expression1, expression2)

The schema is SYSIBM. (The SYSFUN version of the POWER function continues to be available.)

The POWER function returns the result of raising the first argument to the power of the second argument.

The arguments can be of any built-in numeric data type. DECIMAL and REAL arguments are converted to a double-precision floating-point number. If either argument is decimal floating-point, the arguments are converted to DECFLOAT(34) for processing by the function.

The result of the function is:
- INTEGER if both arguments are INTEGER or SMALLINT
- BIGINT if one argument is BIGINT and the other argument is BIGINT, INTEGER or SMALLINT
- DECFLOAT(34) if one of the arguments is decimal floating-point. If either argument is a DECFLOAT and one of the following statements is true, the result is NaN and an invalid operation condition:
  - Both arguments are zero
  - The second argument has a nonzero fractional part
  - The second argument has more than 9 digits
  - The second argument is INFINITY
- DOUBLE otherwise

If the argument is a special decimal floating-point value, the rules for general arithmetic operations for decimal floating-point apply. See “General arithmetic operation rules for decimal floating-point” in page 223.

The result can be null; if any argument is null, the result is the null value.

Example:
- Assume that the host variable HPOWER is an integer with a value of 3.

```VALUES POWER(2, :HPOWER)

Returns the value 8.
The schema is SYSIBM.

The QUANTIZE function returns a decimal floating-point value that is equal in value (except for any rounding) and sign to numeric-expression and that has an exponent equal to the exponent of exp-expression. The number of digits (16 or 34) is the same as the number of digits in numeric-expression.

numeric-expression
An expression that returns a value of any built-in numeric data type. If the argument is not a decimal floating-point value, it is converted to DECFLOAT(34) for processing.

exp-expression
An expression that returns a value of any built-in numeric data type. If the argument is not a decimal floating-point value, it is converted to DECFLOAT(34) for processing. The exp-expression is used as an example pattern for rescaling numeric-expression. The sign and coefficient of exp-expression are ignored.

The coefficient of the result is derived from that of numeric-expression. It is rounded, if necessary (if the exponent is being increased), multiplied by a power of ten (if the exponent is being decreased), or remains unchanged (if the exponent is already equal to that of exp-expression).

The CURRENT DECFLOAT ROUNding MODE special register determines the rounding mode.

Unlike other arithmetic operations on the decimal floating-point data type, if the length of the coefficient after the quantize operation is greater than the precision specified by exp-expression, the result is NaN and a warning is returned (SQLSTATE 0168D). This ensures that, unless there is a warning condition, the exponent of the result of QUANTIZE is always equal to that of exp-expression.

• if either argument is NaN, NaN is returned
• if either argument is sNaN, NaN is returned and a warning is returned (SQLSTATE 01565)
• if both arguments are infinity (positive or negative), infinity with the same sign as the first argument is returned
• if one argument is infinity (positive or negative) and the other argument is not infinity, NaN is returned and a warning is returned (SQLSTATE 0168D)

The result of the function is a DECFLOAT(16) value if both arguments are DECFLOAT(16). Otherwise, the result of the function is a DECFLOAT(34) value. The result can be null; if any argument is null, the result is the null value.

Examples:

• The following examples show the values that are returned by the QUANTIZE function given a variety of input decimal floating-point values and assuming a rounding mode of ROUND_HALF_UP:
QUANTIZE

\[
\begin{align*}
&\text{QUANTIZE}(2.17, \text{DECFLOAT}(0.001)) = 2.170 \\
&\text{QUANTIZE}(2.17, \text{DECFLOAT}(0.01)) = 2.17 \\
&\text{QUANTIZE}(2.17, \text{DECFLOAT}(0.1)) = 2.2 \\
&\text{QUANTIZE}(2.17, \text{DECFLOAT}(\text{1E+0})) = 2 \\
&\text{QUANTIZE}(2.17, \text{DECFLOAT}(\text{1E+1})) = 0E+1 \\
&\text{QUANTIZE}(2, \text{DECFLOAT}(\text{INFINITY})) = \text{NaN} \quad \text{-- warning} \\
&\text{QUANTIZE}(0, \text{DECFLOAT}(\text{1E+5})) = 0E+5 \\
&\text{QUANTIZE}(217, \text{DECFLOAT}(\text{1E-1})) = 217.0 \\
&\text{QUANTIZE}(217, \text{DECFLOAT}(\text{1E+0})) = 217 \\
&\text{QUANTIZE}(217, \text{DECFLOAT}(\text{1E+1})) = 2.2E+2 \\
&\text{QUANTIZE}(217, \text{DECFLOAT}(\text{1E+2})) = 2E+2
\end{align*}
\]

- In the following example the value -0 is returned for the QUANTIZE function. The CHAR function is used to avoid the potential removal of the minus sign by a client program:

\[
\text{CHAR(QUANTIZE}(-0.1, \text{DECFLOAT}(1))) = -0
\]
The schema is SYSFUN.

Returns an integer value in the range 1 to 4, representing the quarter of the year for the date specified in the argument.

The argument must be a DATE, TIMESTAMP, or a valid character string representation of a date or timestamp that is not a CLOB. In a Unicode database, if a supplied argument is a graphic string (except DBCLOB), it is first converted to a character string before the function is executed.

The result of the function is INTEGER. The result can be null; if the argument is null, the result is the null value.
The schema is SYSIBM. (The SYSFUN version of the RADIANS function continues to be available.)

The RADIANS function returns the number of radians for an argument that is expressed in degrees.

The argument can be any built-in numeric data type. If the argument is decimal floating-point, the operation is performed in decimal floating-point; otherwise, the argument is converted to double-precision floating-point for processing by the function.

If the argument is DECFLOAT(n), the result is DECFLOAT(n); otherwise, the result is a double-precision floating-point number. The result can be null; if the argument is null, the result is the null value.

Example:
- Assume that host variable HDEG is an INTEGER with a value of 180. The following statement:
  
  `VALUES RADIANS(:HDEG)`

  Returns the value +3.14159265358979E+000.
The schema is SYSIBM.

The RAISE_ERROR function causes the statement that includes the function to return an error with the specified SQLSTATE, SQLCODE -438, and diagnostic-string.

The RAISE_ERROR function always returns NULL with an undefined data type. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

sqlstate

A character string containing exactly 5 bytes. It must be of type CHAR defined with a length of 5 or type VARCHAR defined with a length of 5 or greater. The sqlstate value must follow the rules for application-defined SQLSTATEs as follows:

* Each character must be from the set of digits ('0' through '9') or non-accented upper case letters ('A' through 'Z')
* The SQLSTATE class (first two characters) cannot be '00', '01' or '02' since these are not error classes.
* If the SQLSTATE class (first two characters) starts with the character '0' through '6' or 'A' through 'H', then the subclass (last three characters) must start with a letter in the range 'I' through 'Z'.
* If the SQLSTATE class (first two characters) starts with the character '7', '8', '9' or 'I' through 'Z', then the subclass (last three characters) can be any of '0' through '9' or 'A' through 'Z'.

If the SQLSTATE does not conform to these rules an error occurs (SQLSTATE 428B3).

diagnostic-string

An expression of type CHAR or VARCHAR that returns a character string of up to 70 bytes that describes the error condition. If the string is longer than 70 bytes, it will be truncated.

To use this function in a context where the data type cannot be determined, a cast specification must be used to give the null returned value a data type. A CASE expression is where the RAISE_ERROR function will be most useful.

Example:

List employee numbers and education levels as Post Graduate, Graduate and Diploma. If an education level is greater than 20, raise an error.

```sql
SELECT EMPNO,
CASE WHEN EDUCLVL < 16 THEN 'Diploma'
WHEN EDUCLVL < 18 THEN 'Graduate'
WHEN EDUCLVL < 21 THEN 'Post Graduate'
ELSE RAISE_ERROR('70001',
    'EDUCLVL has a value greater than 20')
END
FROM EMPLOYEE
```
The schema is SYSFUN.

The RAND function returns a floating point value between 0 and 1.

If an expression is specified, it is used as the seed value. The expression must be a built-in SMALLINT or INTEGER data type with a value between 0 and 2,147,483,647.

The data type of the result is double-precision floating point. If the argument is null, the result is the null value.

A specific seed value will produce the same sequence of random numbers for a specific instance of a RAND function in a query each time the query is executed. If a seed value is not specified, a different sequence of random numbers is produced each time the query is executed within the same session. To produce a set of random numbers that varies from session to session, specify a random seed; for example, one that is based on the current time.

RAND is a non-deterministic function.
The REAL function returns a single-precision floating-point representation of either:

- A number
- A string representation of a number

### Numeric to real

**numeric-expression**

An expression that returns a value of any built-in numeric data type.

The result is the same number that would occur if the argument were assigned to a single-precision floating-point column or variable. If the numeric value of the argument is not within the range of single-precision floating-point, an error is returned (SQLSTATE 22003).

### String to real

**string-expression**

An expression that returns a value that is character-string or Unicode graphic-string representation of a number. The data type of string-expression must not be a CLOB or a DBCLOB (SQLSTATE 42884).

The result is the same number that would result from CAST(string-expression AS REAL). Leading and trailing blanks are eliminated and the resulting string must conform to the rules for forming a valid numeric constant (SQLSTATE 22018). If the numeric value of the argument is not within the range of single-precision floating-point, an error is returned (SQLSTATE 22003).

The result of the function is a single-precision floating-point number. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

**Note:** The CAST specification should be used to increase the portability of applications. For more information, see “CAST specification”.

Example:

Using the EMPLOYEE table, find the ratio of salary to commission for employees whose commission is not zero. The columns involved (SALARY and COMM) have DECIMAL data types. The result is desired in single-precision floating point. Therefore, REAL is applied to SALARY so that the division is carried out in floating point (actually double-precision) and then REAL is applied to the complete expression to return the result in single-precision floating point.
SELECT EMPNO, REAL(REAL(SALARY)/COMM)
FROM EMPLOYEE
WHERE COMM > 0
The schema is SYSIBM.

The REC2XML function returns a string formatted with XML tags, containing column names and column data. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

**decimal-constant**
The expansion factor for replacing column data characters. The decimal value must be greater than 0.0 and less than or equal to 6.0. (SQLSTATE 42820).

The *decimal-constant* value is used to calculate the result length of the function. For every column with a character data type, the length attribute of the column is multiplied by this expansion factor before it is added in to the result length.

To specify no expansion, use a value of 1.0. Specifying a value less than 1.0 reduces the calculated result length. If the actual length of the result string is greater than the calculated result length of the function, then an error is raised (SQLSTATE 22001).

**format-string**
The string constant that specifies which format the function is to use during execution.

The *format-string* is case-sensitive, so the following values must be specified in uppercase to be recognized.

**COLATTVAL** or **COLATTVAL XML**
These formats return a string with columns as attribute values.

Column names may or may not be valid XML attribute values. For column names which are not valid XML attribute values, character replacement is performed on the column name before it is included in the result string.

Column values may or may not be valid XML element names. If the *format-string* **COLATTVAL** is specified, then for the column names which are
not valid XML element values, character replacement is performed on the
column value before it is included in the result string. If the format-string
COLATTVAL XML is specified, then character replacement is not performed
on column values (although character replacement is still performed on
column names).

row-tag-string
A string constant that specifies the tag used for each row. If an empty string is
specified, then a value of 'row' is assumed.

If a string of one or more blank characters is specified, then no beginning
row-tag-string or ending row-tag-string (including the angle bracket delimiters)
will appear in the result string.

column-name
A qualified or unqualified name of a table column. The column must have one
of the following data types (SQLSTATE 42815):
• numeric (SMALLINT, INTEGER, BIGINT, DECIMAL, REAL, DOUBLE)
• character string (CHAR, VARCHAR; a character string with a subtype of BIT
  DATA is not allowed)
• datetime (DATE, TIME, TIMESTAMP)
• a user-defined type based on one of the above types

The same column name cannot be specified more than once (SQLSTATE
42734).

The result of the function is VARCHAR. The maximum length is 32 672 bytes
(SQLSTATE 54006).

Consider the following invocation:

REC2XML (dc, fs, rt, c1, c2, ..., cn)

If the value of "fs" is either "COLATTVAL" or "COLATTVAL XML", then the result
is the same as this expression:

'<' CONCAT rt CONCAT '>' CONCAT y1 CONCAT y2
CONCAT ... CONCAT yn CONCAT '</' CONCAT rt CONCAT '>

where yn is equivalent to:

'<column name="" CONCAT xvcn CONCAT vn

and vn is equivalent to:

'" CONCAT r_n CONCAT '</column>'

if the column is not null, and

'" null="true"/>

if the column value is null.

xvcn is equivalent to a string representation of the column name of cn, where any
characters appearing in Table 55 on page 516 are replaced with the corresponding
representation. This ensures that the resulting string is a valid XML attribute or
element value token.
The $r_n$ is equivalent to a string representation as indicated in Table 54.

### Table 54. Column Values String Result

<table>
<thead>
<tr>
<th>Data type of $c_n$</th>
<th>$r_n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR, VARCHAR</td>
<td>The value is a string. If the <code>format-string</code> does not end in the characters &quot;_XML&quot;, then each character in $c_n$ is replaced with the corresponding replacement representation from Table 55 as indicated. The length attribute is: $dc \times$ the length attribute of $c_n$.</td>
</tr>
<tr>
<td>SMALLINT, INTEGER, BIGINT, DECIMAL, NUMERIC, REAL, DOUBLE</td>
<td>The value is <code>LTRIM(RTRIM(CHAR($c_n$)))</code>. The length attribute is the result length of CHAR($c_n$). The decimal character is always the period (\text{.}) character.</td>
</tr>
<tr>
<td>DATE</td>
<td>The value is CHAR($c_n$ISO). The length attribute is the result length of CHAR($c_n$ISO).</td>
</tr>
<tr>
<td>TIME</td>
<td>The value is CHAR($c_n$JIS). The length attribute is the result length of CHAR($c_n$JIS).</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>The value is CHAR($c_n$). The length attribute is the result length of CHAR($c_n$).</td>
</tr>
</tbody>
</table>

**Character replacement:**

Depending on the value specified for the `format-string`, certain characters in column names and column values will be replaced to ensure that the column names form valid XML attribute values and the column values form valid XML element values.

### Table 55. Character Replacements for XML Attribute Values and Element Values

<table>
<thead>
<tr>
<th>Character</th>
<th>Replacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>&lt;</td>
</tr>
<tr>
<td>&gt;</td>
<td>&gt;</td>
</tr>
<tr>
<td>&quot;</td>
<td>&quot;</td>
</tr>
<tr>
<td>*</td>
<td>&amp;</td>
</tr>
<tr>
<td>&amp;</td>
<td>'</td>
</tr>
</tbody>
</table>

**Examples:**

**Note:** REC2XML does not insert blank spaces or new line characters in the output. All example output has been formatted to enhance readability.

- Using the DEPARTMENT table in the sample database, format the department table row, except the DEPTNAME and LOCATION columns, for department 'D01' into an XML string. Since the data does not contain any of the characters which require replacement, the expansion factor will be 1.0 (no expansion). Also note that the MGRNO value is null for this row.

```sql
SELECT REC2XML (1.0, 'COLATTVAL', '', DEPTNO, MGRNO, ADMRDEPT)
FROM DEPARTMENT
WHERE DEPTNO = 'D01'
```

This example returns the following VARCHAR(117) string:
A 5-day university schedule introduces a class named ‘&43<FIE’ to a table called CL_SCHED, with a new format for the CLASS_CODE column. Using the REC2XML function, this example formats an XML string with this new class data, except for the class end time.

The length attribute for the REC2XML call (see below) with an expansion factor of 1.0 would be 128 (11 for the ‘<row>’ and ‘</row>’ overhead, 21 for the column names, 75 for the ‘<column name=’, ‘>’, ‘</column>’ and double quotes, 7 for the CLASS_CODE data, 6 for the DAY data, and 8 for the STARTING data). Since the ‘&’ and ‘<’ characters will be replaced, an expansion factor of 1.0 will not be sufficient. The length attribute of the function will need to support an increase from 7 to 14 bytes for the new format CLASS_CODE data.

However, since it is known that the DAY value will never be more than 1 digit long, an unused extra 5 units of length are added to the total. Therefore, the expansion only needs to handle an increase of 2. Since CLASS_CODE is the only character string column in the argument list, this is the only column data to which the expansion factor applies. To get an increase of 2 for the length, an expansion factor of 9/7 (approximately 1.2857) would be needed. An expansion factor of 1.3 will be used.

```
SELECT REC2XML (1.3, 'COLATTVAL', 'record', CLASS_CODE, DAY, STARTING)
FROM CL_SCHED
WHERE CLASS_CODE = '&43<FIE'
```

This example returns the following VARCHAR(167) string:

```
<record>
  <column name="CLASS_CODE">&amp;43&lt;FIE</column>
  <column name="DAY">5</column>
  <column name="STARTING">06:45:00</column>
</record>
```

Assume that new rows have been added to the EMP_RESUME table in the sample database. The new rows store the resumes as strings of valid XML. The COLATTVAL_XML format-string is used so character replacement will not be carried out. None of the resumes are more than 3500 bytes in length. The following query is used to select the XML version of the resumes from the EMP_RESUME table and format it into an XML document fragment.

```
SELECT REC2XML (1.0, 'COLATTVAL_XML', 'row', EMPNO, RESUME_XML)
FROM (SELECT EMPNO, CAST(RESUME AS VARCHAR(3500)) AS RESUME_XML
  FROM EMP_RESUME
  WHERE RESUME_FORMAT = 'XML') AS EMP_RESUME_XML
```

This example returns a row for each employee who has a resume in XML format. Each returned row will be a string with the following format:

```
<row>
  <column name="EMPNO">{employee number}</column>
  <column name="RESUME_XML">{resume in XML}</column>
</row>
```

Where "{employee number}" is the actual EMPNO value for the column and "{resume in XML}" is the actual XML fragment string value that is the resume.
REPEAT

The schema is SYSFUN.

Returns a character string composed of the first argument repeated the number of
times specified by the second argument. In a Unicode database, if a supplied
argument is a graphic string, it is first converted to a character string before the
function is executed.

The first argument is a character string or binary string type. For a VARCHAR the
maximum length is 4000 bytes and for a CLOB or a binary string the maximum
length is 1 048 576 bytes. The second argument can be SMALLINT or INTEGER.

The result of the function is:
• VARCHAR(4000) if the first argument is VARCHAR (not exceeding 4000 bytes)
or CHAR
• CLOB(1M) if the first argument is CLOB.
• BLOB(1M) if the first argument is BLOB.

The result can be null; if any argument is null, the result is the null value.

Example:
• List the phrase ‘REPEAT THIS’ five times.

   VALUES CHAR(REPEAT(‘REPEAT THIS’, 5), 60)

This example return the following:

<table>
<thead>
<tr>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>REPEAT THIS REPEAT THIS REPEAT THIS REPEAT THIS REPEAT THIS</td>
</tr>
</tbody>
</table>

As mentioned, the output of the REPEAT function is VARCHAR(4000). For this
eample, the CHAR function has been used to limit the output of REPEAT to 60
bytes.
The schema is SYSIBM. The SYSFUN version of the REPLACE function continues to be available but it is not sensitive to the database collation.

Replaces all occurrences of search-string in source-string with replace-string. If search-string is not found in source-string, search-string is returned unchanged. The search is done using the collation of the database unless source-string, search-string or replace-string is defined as FOR BIT DATA, in which case the search is done using a binary comparison.

source-string
An expression that specifies the source string. The expression must return a value that is a built-in CHAR, VARCHAR, GRAPHIC, VARGRAPHIC, numeric, or datetime data type. If the value is not a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type, it is implicitly cast to VARCHAR before evaluating the function.

search-string
An expression that specifies the string to be removed from the source string. The expression must return a value that is a built-in CHAR, VARCHAR, GRAPHIC, VARGRAPHIC, numeric, or datetime data type. If the value is not a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type, it is implicitly cast to VARCHAR before evaluating the function.

replace-string
An expression that specifies the replacement string. The expression must return a value that is a built-in CHAR, VARCHAR, GRAPHIC, VARGRAPHIC, numeric, or datetime data type. If the value is not a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type, it is implicitly cast to VARCHAR before evaluating the function.

The actual length of each string must be 32,672 bytes or less for character strings, or 16,336 or less for graphic strings. All three arguments must have compatible data types.

If source-string, search-string or replace-string is defined as FOR BIT DATA, the result is VARCHAR FOR BIT DATA. If source-string is a character string, the result is VARCHAR. If source-string is a graphic string, the result is VARGRAPHIC. If one argument is character FOR BIT DATA, the other arguments must not be graphic (SQLSTATE 42846).

The length attribute of the result depends on the arguments:
• If the length attribute of replace-string is less than or equal to the length attribute of search-string, the length attribute of the result is the length attribute of source-string.
• If the length attribute of replace-string is greater than the length attribute of search-string, the length attribute of the result is determined as follows, depending on the data type of the result:
  – For VARCHAR:
    - If $L1 \leq 4000$, the length attribute of the result is $\min(4000, (L3 \times (L1/L2)) + \mod(L1,L2))$
REPLACE

- Otherwise, the length attribute of the result is $\min(32672, (L3 \times (L1/L2)) + \mod(L1, L2))$
- For VARGRAPHIC:
  - If $L1 \leq 2000$, the length attribute of the result is $\min(2000, (L3 \times (L1/L2)) + \mod(L1, L2))$
  - Otherwise, the length attribute of the result is $\min(16336, (L3 \times (L1/L2)) + \mod(L1, L2))$

where:
- $L1$ is the length attribute of source-string
- $L2$ is the length attribute of the search-string if the search string is a string constant. Otherwise, $L2$ is 1.
- $L3$ is the length attribute of replace-string

If the result is a character string, the length attribute of the result must not exceed 32,672. If the result is a graphic string, the length attribute of the result must not exceed 16,336.

The actual length of the result is the actual length of source-string plus the number of occurrences of search-string that exist in source-string multiplied by the actual length of replace-string minus the actual length of search-string.

If the actual length of the replace-string exceeds the maximum for the return data type, an error is returned. If any argument can be null, the result can be null; if any argument is null, the result is the null value.

Examples:
- Replace all occurrences of the letter 'N' in the word 'DINING' with 'VID'.
  
  VALUES CHAR (REPLACE ('DINING', 'N', 'VID'), 10)

  The result is the string 'DIVIDIVIDG'.
- In a Unicode database with case-insensitive collation UCA500R1_LEN_S1, replace the word 'QUICK' with the word 'LARGE'.
  VALUES REPLACE ('The quick brown fox', 'QUICK', 'LARGE')

  The result is the string 'The LARGE brown fox'.
The schema is SYSIBM. The function name cannot be specified as a qualified name.

The RID_BIT and RID functions return the row identifier (RID) of a row in different formats. The RID is used to uniquely identify a row. Each function might return different values when it is invoked multiple times against a row. For example, after the reorg utility is run against a table, the RID_BIT and RID functions might return different values for a row than would have been returned prior to running the utility. The RID_BIT and RID functions are non-deterministic. The RID_BIT function result, unlike the RID function, contains table information to protect from inadvertently using it with a different table. The RID function is not supported in a partitioned database environment.

table-designator

Uniquely identifies a base table, view, or nested table expression (SQLSTATE 42703). If `table-designator` specifies a view or nested table expression, the RID_BIT and RID functions return the RID of the base table of the view or nested table expression. The specified view or nested table expression must contain only one base table in its outer subselect (SQLSTATE 42703). The `table-designator` must be deletable (SQLSTATE 42703). For information about deletable views, see the “Notes” section of “CREATE VIEW”.

If `table-designator` is not specified, the FROM clause must contain only one element which can be derived to be the table designator (SQLSTATE 42703).

The result of the RID_BIT function is VARCHAR (16) FOR BIT DATA. The result can be null. The result of the RID function is BIGINT. The result can be null.

Notes:

- To implement optimistic locking in an application, use the values returned by the ROW CHANGE TOKEN expression as arguments to the RID_BIT scalar function.
- **Compatibilities**: The following syntax is supported, but is non-standard and should not be used:
  - The pseudo column “ROWID” can be used to refer to the RID. An unqualified ROWID reference is equivalent to RID_BIT() and a qualified ROWID such as EMPLOYEE.ROWID is equivalent to RID_BIT(EMPLOYEE).

Examples:

- Return the RID and last name of employees in department 20 from the EMPLOYEE table.
  
  ```sql
  SELECT RID_BIT(EMPLOYEE), ROW CHANGE TOKEN FOR EMPLOYEE, LASTNAME
  FROM EMPLOYEE
  WHERE DEPTNO = '20'
  ```

- Given table EMP1, which is defined as follows:
  
  ```sql
  CREATE TABLE EMP1 (  
    EMPNO CHAR(6),  
    NAME  CHAR(30),  
  ```
RID_BIT and RID

SET host variable HV_EMP_RID to the value of the RID_BIT built-in scalar function, and HV_EMP_RCT to the value of the ROW CHANGE TOKEN expression for the row corresponding to employee number 3500.

```
SELECT RID_BIT(EMP1), ROW CHANGE TOKEN FOR EMP1 INTO :HV_EMP_RID, :HV_EMP_RCT FROM EMP1 WHERE EMPNO = '3500'
```

Using that RID value to identify the employee, and user-defined function UPDATE_RESUME, increase the employee’s salary by $1000 and update the employee’s resume.

```
UPDATE EMP1 SET
  SALARY = SALARY + 1000,
  RESUME = UPDATE_RESUME(:HV_RESUME)
WHERE RID_BIT(EMP1) = :HV_EMP_RID
AND ROW CHANGE TOKEN FOR EMP1 = :HV_EMP_RCT
```
The schema is SYSIBM. The SYSFUN version of the RIGHT function continues to be available.

The RIGHT function returns the rightmost string of string-expression of length length, expressed in the specified string unit. If string-expression is a character string, the result is a character string. If string-expression is a graphic string, the result is a graphic string.

string-expression
An expression that specifies the string from which the result is derived. The expression must return a value that is a built-in string, numeric, or datetime data type. If the value is not a string data type, it is implicitly cast to VARCHAR before evaluating the function. A substring of string-expression is zero or more contiguous code points of string-expression.

length
An expression that specifies the length of the result. The expression must return a value that is a built-in numeric, CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type. If the value is not of type INTEGER, it is implicitly cast to INTEGER before evaluating the function. The value must be between 0 and the length of string-expression, expressed in units that are either implicitly or explicitly specified (SQLSTATE 22011). If OCTETS is specified and the result is graphic data, the value must be an even number between 0 and twice the length attribute of string-expression (SQLSTATE 428GC).

CODEUNITS16, CODEUNITS32, or OCTETS
Specifies the string unit of length.

CODEUNITS16 specifies that length is expressed in 16-bit UTF-16 code units.
CODEUNITS32 specifies that length is expressed in 32-bit UTF-32 code units.
OCTETS specifies that length is expressed in bytes.

If the string unit is specified as CODEUNITS16 or CODEUNITS32, and string-expression is a binary string or bit data, an error is returned (SQLSTATE 428GC). If the string unit is specified as OCTETS and string-expression is a graphic string, length must be an even number; otherwise, an error is returned (SQLSTATE 428GC). If a string unit is not explicitly specified, the data type of the result determines the unit that is used. If the result is graphic data, length is expressed in two-byte units; otherwise, it is expressed in bytes. For more information about CODEUNITS16, CODEUNITS32, and OCTETS, see “String units in built-in functions” in “Character strings”.

The string-expression is padded on the right with the necessary number of padding characters so that the specified substring of string-expression always exists. The character used for padding is the same character that is used to pad the string in contexts where padding would occur. For more information on padding, see “String assignments” in “Assignments and comparisons”.

Chapter 3. Functions 523
The result of the function is a varying-length string with a length attribute that is the same as the length attribute of \textit{string-expression} and a data type that depends on the data type of \textit{string-expression}:

- \textbf{VARCHAR} if \textit{string-expression} is CHAR or VARCHAR
- \textbf{CLOB} if \textit{string-expression} is CLOB
- \textbf{VARGRAPHIC} if \textit{string-expression} is GRAPHIC or VARGRAPHIC
- \textbf{DBCLOB} if \textit{string-expression} is DBCLOB
- \textbf{BLOB} if \textit{string-expression} is BLOB

The actual length of the result (in string units) is \textit{length}.

If any argument can be null, the result can be null; if any argument is null, the result is the null value.

Examples:

- Assume that variable ALPHA has a value of 'ABCDEF'. The following statement:
  \begin{verbatim}
  SELECT RIGHT(ALPHA,3)
  FROM SYSIBM.SYSDUMMY1
  \end{verbatim}

  returns 'DEF', which are the three rightmost characters in ALPHA.

- Assume that variable NAME, which is defined as VARCHAR(50), has a value of 'KATIE AUSTIN', and that the integer variable LASTNAME_LEN has a value of 6. The following statement:
  \begin{verbatim}
  SELECT RIGHT(NAME,LASTNAME_LEN)
  FROM SYSIBM.SYSDUMMY1
  \end{verbatim}

  returns the value 'AUSTIN'.

- The following statement returns a zero-length string.
  \begin{verbatim}
  SELECT RIGHT('ABCABC',0)
  FROM SYSIBM.SYSDUMMY1
  \end{verbatim}

- The FIRSTNME column in the EMPLOYEE table is defined as VARCHAR(12). Find the first name of an employee whose last name is 'BROWN' and return the first name in a 10-byte string.
  \begin{verbatim}
  SELECT RIGHT(FIRSTNME, 10)
  FROM EMPLOYEE
  WHERE LASTNAME = 'BROWN'
  \end{verbatim}

  returns a VARCHAR(12) string that has the value 'DAVID' followed by five blank characters.

- In a Unicode database, FIRSTNAME is a VARCHAR(12) column. One of its values is the 6-character string 'Jürgen'. When FIRSTNAME has this value:

  Function... Returns...

  \begin{verbatim}
  RIGHT(FIRSTNAME,5,CODEUNITS32) 'ürgen' -- x'C3BC7267656E'
  RIGHT(FIRSTNAME,5,CODEUNITS16) 'ürgen' -- x'C3BC7267656E'
  \end{verbatim}

  The following example works with the Unicode string '&N~AB', where '&' is the musical symbol G clef character, and '~' is the combining tilde character. This string is shown below in different Unicode encoding forms:

<table>
<thead>
<tr>
<th></th>
<th>'G'</th>
<th>'N'</th>
<th>'~'</th>
<th>'A'</th>
<th>'B'</th>
</tr>
</thead>
<tbody>
<tr>
<td>UTF-8</td>
<td>x'F09D849E'</td>
<td>x'4E'</td>
<td>x'CC83'</td>
<td>x'41'</td>
<td>x'42'</td>
</tr>
<tr>
<td>UTF-16BE</td>
<td>x'D834DD1E'</td>
<td>x'004E'</td>
<td>x'0303'</td>
<td>x'0041'</td>
<td>x'0042'</td>
</tr>
</tbody>
</table>
Assume that the variable UTF8_VAR, with a length attribute of 20 bytes, contains the UTF-8 representation of the string.

```
SELECT RIGHT(UTF8_VAR, 2, CODEUNITS16),
       RIGHT(UTF8_VAR, 2, CODEUNITS32),
       RIGHT(UTF8_VAR, 2, OCTETS)
FROM SYSIBM.SYSDUMMY1
```
returns the values 'AB', 'AB', and 'AB', respectively.

```
SELECT RIGHT(UTF8_VAR, 5, CODEUNITS16),
       RIGHT(UTF8_VAR, 5, CODEUNITS32),
       RIGHT(UTF8_VAR, 5, OCTETS)
FROM SYSIBM.SYSDUMMY1
```
returns the values '?N~AB', '&N~AB', and 'N~AB', respectively, where '?' is X'EDB49E'.

```
SELECT RIGHT(UTF8_VAR, 10, CODEUNITS16),
       RIGHT(UTF8_VAR, 10, CODEUNITS32),
       RIGHT(UTF8_VAR, 10, OCTETS)
FROM SYSIBM.SYSDUMMY1
```
returns the values '&N~ABbbbb', '&N~ABbbbbb', and '&N~ABb', respectively, where 'b' represents the blank character.

Assume that the variable UTF16_VAR, with a length attribute of 20 code units, contains the UTF-16BE representation of the string.

```
SELECT RIGHT(UTF16_VAR, 2, CODEUNITS16),
       RIGHT(UTF16_VAR, 2, CODEUNITS32),
       RIGHT(UTF16_VAR, 2, OCTETS)
FROM SYSIBM.SYSDUMMY1
```
returns the values 'AB', 'AB', and 'B', respectively.

```
SELECT RIGHT(UTF16_VAR, 5, CODEUNITS16),
       RIGHT(UTF16_VAR, 5, CODEUNITS32),
       RIGHT(UTF16_VAR, 6, OCTETS)
FROM SYSIBM.SYSDUMMY1
```
returns the values '?N~AB', '&N~AB', and '~AB', respectively, where '?' is the standalone low surrogate X'DD1E'.

```
SELECT RIGHT(UTF16_VAR, 10, CODEUNITS16),
       RIGHT(UTF16_VAR, 10, CODEUNITS32),
       RIGHT(UTF16_VAR, 10, OCTETS)
FROM SYSIBM.SYSDUMMY1
```
returns the values '&N~ABbbbb', '&N~ABbbbbb', and '?N~AB', respectively, where 'b' represents the blank character and '?' is X'DD1E'.
ROUND

ROUND numeric:

```
ROUND((numeric-expression1, numeric-expression2))
```

ROUND datetime:

```
ROUND((datetime-expression, format-string, locale-name))
```

The schema is SYSIBM. The SYSFUN version of the ROUND numeric function continues to be available.

The ROUND function returns a rounded value of:
- A number, rounded to the specified number of places to the right or left of the decimal point, if the result of the first argument is a numeric value
- A datetime value, rounded to the unit specified by `format-string`, if the first argument is a DATE, TIME, or TIMESTAMP

### ROUND numeric

If `numeric-expression1` is positive, a digit value of 5 or greater is an indication to round to the next higher positive number. For example, `ROUND(3.5, 0) = 4`. If `numeric-expression1` is negative, a digit value of 5 or greater is an indication to round to the next lower negative number. For example, `ROUND(-3.5, 0) = -4`.

#### `numeric-expression1`

An expression that must return a value that is a built-in CHAR, VARCHAR, GRAPHIC, VARGRAPHIC, or numeric data type. If the value is not a numeric data type, it is implicitly cast to `DECFLOAT(34)` before evaluating the function.

If the expression is a decimal floating-point data type, the `DECFLOAT` rounding mode will not be used. The rounding behavior of `ROUND` corresponds to a value of `ROUND_HALF_UP`. If a different rounding behavior is wanted, use the `QUANTIZE` function.

#### `numeric-expression2`

An expression that returns a value that is a built-in numeric data type. If the value is not of type INTEGER, it is implicitly cast to INTEGER before evaluating the function.

If `numeric-expression2` is not negative, `numeric-expression1` is rounded to the absolute value of `numeric-expression2` number of places to the right of the decimal point.

If `numeric-expression2` is negative, `numeric-expression1` is rounded to the absolute value of `numeric-expression2+1` number of places to the left of the decimal point.

If the absolute value of a negative `numeric-expression2` is larger than the number of digits to the left of the decimal point, the result is 0.
example, \( \text{ROUND}(748.58,-4) = 0 \). If \( \text{numeric-expression1} \) is positive, a
digit value of 5 is rounded to the next higher positive number. If
\( \text{numeric-expression1} \) is negative, a digit value of 5 is rounded to the next
lower negative number.

The data type and length attribute of the result are the same as the data
type and length attribute of the first argument, except that the precision is
increased by one if the \( \text{numeric-expression1} \) is DECIMAL and the precision
is less than 31. For example, an argument with a data type of
\( \text{DECIMAL}(5,2) \) results in \( \text{DECIMAL}(6,2) \). An argument with a data type of
\( \text{DECIMAL}(31,2) \) results in \( \text{DECIMAL}(31,2) \). The scale is the same as the
scale of the first argument.

If either argument can be null or if the argument is not a decimal floating-point number and the database is configured with
dft_sqlmathwarn set to YES, the result can be null. If either argument is
null, the result is the null value.

This function is not affected by the setting of the CURRENT DECFLOAT ROUNDBING MODE special register, even for decimal floating-point
arguments. The rounding behavior of \text{ROUND} corresponds to a value of
\text{ROUND_HALF_UP}. If you want behavior for a decimal floating-point value that conforms to the rounding mode specified by the CURRENT
DECFLOAT ROUNDBING MODE special register, use the \text{QUANTIZE}
function instead.

\text{ROUND} \text{datetime}

If \( \text{datetime-expression} \) has a datetime data type, the \text{ROUND} function returns
\( \text{datetime-expression} \) rounded to the unit specified by the \text{format-string}. If
\text{format-string} is not specified, \( \text{datetime-expression} \) is rounded to the nearest
day, as if ‘DD’ is specified for \text{format-string}.

\text{datetime-expression}

An expression that must return a value that is a date, a time, or a
timestamp. String representations of these data types are not supported
and must be explicitly cast to a DATE, TIME, or TIMESTAMP for use
with this function; alternatively, you can use the \text{ROUND_TIMESTAMP}
function for a string representation of a date or timestamp.

\text{format-string}

An expression that returns a built-in character string data type with an
actual length that is not greater than 254 bytes. The format element in
\text{format-string} specifies how \( \text{datetime-expression} \) should be rounded. For
example, if \text{format-string} is ‘DD’, a timestamp that is represented by
\( \text{datetime-expression} \) is rounded to the nearest day. Leading and trailing
blanks are removed from the string, and the resulting substring must
be a valid format element for the type of \( \text{datetime-expression} \) (SQLSTATE
22007). The default is ‘DD’, which cannot be used if the data type of
\( \text{datetime-expression} \) is TIME.

Allowable values for \text{format-string} are listed in the table of format
elements listed below.

\text{locale-name}

A character constant that specifies the locale used to determine the first
day of the week when using format element values DAY, DY, or D. The
value of \text{locale-name} is not case sensitive and must be a valid locale
(SQLSTATE 42815). For information on valid locales and their naming,
see “Locale names for SQL and XQuery”. If `locale-name` is not specified, the value of the special register CURRENT LOCALE LC_TIME is used.

The result of the function has the same DATE type as `datetime-expression`. The result can be null; if any argument is null, the result is the null value.

The following format elements are used to identify the rounding or truncation unit of the datetime value in the ROUND, ROUND_TIMESTAMP, TRUNCATE and TRUNC_TIMESTAMP functions.

**Table 56. Format elements for ROUND, ROUND_TIMESTAMP, TRUNCATE, and TRUNC_TIMESTAMP**

<table>
<thead>
<tr>
<th>Format element</th>
<th>Rounding or truncating unit</th>
<th>ROUND example</th>
<th>TRUNCATE example</th>
</tr>
</thead>
<tbody>
<tr>
<td>CC SCC</td>
<td>Century</td>
<td>Input Value: 1897-12-04-12.22.22.000000</td>
<td>Input Value: 1897-12-04-12.22.22.000000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Result: 1901-01-01-00.00.00.000000</td>
<td>Result: 1801-01-01-00.00.00.000000</td>
</tr>
<tr>
<td>SYYYY YYYY YEAR SYEAR YYYY YY Y</td>
<td>ISO Year</td>
<td>Input Value: 1897-12-04-12.22.22.000000</td>
<td>Input Value: 1897-12-04-12.22.22.000000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Result: 1898-01-01-00.00.00.000000</td>
<td>Result: 1897-01-04-00.00.00.000000</td>
</tr>
<tr>
<td>IYYYY IYY IY I</td>
<td>ISO Year</td>
<td>Input Value: 1897-12-04-12.22.22.000000</td>
<td>Input Value: 1897-12-04-12.22.22.000000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Result: 1898-01-03-00.00.00.000000</td>
<td>Result: 1897-01-04-00.00.00.000000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Result: 1999-07-01-00.00.00.000000</td>
<td>Result: 1999-04-01-00.00.00.000000</td>
</tr>
</tbody>
</table>
### Table 56. Format elements for ROUND, ROUND_TIMESTAMP, TRUNCATE, and TRUNC_TIMESTAMP (continued)

<table>
<thead>
<tr>
<th>Format element</th>
<th>Rounding or truncating unit</th>
<th>ROUND example</th>
<th>TRUNCATE example</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Rounds up on the 16th day of the month.</td>
<td>Result: 1999-07-01-00.00.00.000000</td>
<td>Result: 1999-06-01-00.00.00.000000</td>
</tr>
<tr>
<td></td>
<td>Not valid for TIME argument.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Rounds up on the 12th hour of the 4th day of the week, with respect to the first day of the year.</td>
<td>Result: 2000-05-06-00.00.00.000000</td>
<td>Result: 2000-04-29-00.00.00.000000</td>
</tr>
<tr>
<td></td>
<td>Not valid for TIME argument.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Rounds up on the 12th hour of the 4th day of the week, with respect to the first day of the ISO year.</td>
<td>Result: 2000-05-08-00.00.00.000000</td>
<td>Result: 2000-05-01-00.00.00.000000</td>
</tr>
<tr>
<td></td>
<td>Not valid for TIME argument.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Rounds up on the 12th hour of the 4th day of the week, with respect to the first day of the month.</td>
<td>Result: 2000-06-22-00.00.00.000000</td>
<td>Result: 2000-06-15-00.00.00.000000</td>
</tr>
<tr>
<td></td>
<td>Not valid for TIME argument.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Rounds up on the 12th hour of the day.</td>
<td>Result: 2000-05-18-00.00.00.000000</td>
<td>Result: 2000-05-17-00.00.00.000000</td>
</tr>
</tbody>
</table>
Table 56. Format elements for ROUND, ROUND_TIMESTAMP, TRUNCATE, and TRUNC_TIMESTAMP (continued)

<table>
<thead>
<tr>
<th>Format element</th>
<th>Rounding or truncating unit</th>
<th>ROUND example</th>
<th>TRUNCATE example</th>
</tr>
</thead>
<tbody>
<tr>
<td>DAY, DY, D</td>
<td>Starting day of the week.</td>
<td>Input Value:</td>
<td>Input Value:</td>
</tr>
<tr>
<td></td>
<td>Rounds up with respect to</td>
<td>2000-05-17-</td>
<td>2000-05-17-</td>
</tr>
<tr>
<td></td>
<td>the 12th hour of the 4th</td>
<td>12.59.59.000000</td>
<td>12.59.59.000000</td>
</tr>
<tr>
<td></td>
<td>day of the week. The first</td>
<td>Result:</td>
<td>Result:</td>
</tr>
<tr>
<td></td>
<td>day of the week is based</td>
<td>2000-05-21-</td>
<td>2000-05-14-</td>
</tr>
<tr>
<td></td>
<td>on the locale (see locale-</td>
<td>00.00.00.000000</td>
<td>00.00.00.000000</td>
</tr>
<tr>
<td></td>
<td>name). Not valid for TIME</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>argument.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HH, HH12, HH24</td>
<td>Hour</td>
<td>Input Value:</td>
<td>Input Value:</td>
</tr>
<tr>
<td></td>
<td>Rounds up at 30 minutes.</td>
<td>2000-05-17-</td>
<td>2000-05-17-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>23.59.59.000000</td>
<td>23.59.59.000000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Result:</td>
<td>Result:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2000-05-18-</td>
<td>2000-05-17-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00.00.00.000000</td>
<td>23.00.00.000000</td>
</tr>
<tr>
<td>MI</td>
<td>Minute</td>
<td>Input Value:</td>
<td>Input Value:</td>
</tr>
<tr>
<td></td>
<td>Rounds up at 30 seconds.</td>
<td>2000-05-17-</td>
<td>2000-05-17-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>23.38.45.000000</td>
<td>23.58.45.000000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Result:</td>
<td>Result:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2000-05-17-</td>
<td>2000-05-17-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>23.59.00.000000</td>
<td>23.58.00.000000</td>
</tr>
<tr>
<td>SS</td>
<td>Second</td>
<td>Input Value:</td>
<td>Input Value:</td>
</tr>
<tr>
<td></td>
<td>Rounds up at half a second.</td>
<td>2000-05-17-</td>
<td>2000-05-17-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>23.38.45.500000</td>
<td>23.58.45.500000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Result:</td>
<td>Result:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2000-05-17-</td>
<td>2000-05-17-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>23.58.46.000000</td>
<td>23.58.45.000000</td>
</tr>
</tbody>
</table>

Note: The format elements in Table 56 on page 528 must be specified in uppercase.

If a format element that applies to a time part of a value is specified for a date argument, the date argument is returned unchanged. If a format element that is not valid for a time argument is specified for a time argument, an error is returned (SQLSTATE 22007).

Notes

- Determinism: ROUND is a deterministic function. However, the following invocations of the function depend on the value of the special register CURRENT LOCALE LC_TIME.
  - Round of a datetime value when locale-name is not explicitly specified and one of the following is true:
    - format-string is not a constant
    - format-string is a constant and includes format elements that are locale sensitive
Invocations of the function that depend on the value of a special register cannot be used wherever special registers cannot be used.

**Examples**

- Calculate the value of 873.726, rounded to 2, 1, 0, -1, -2, -3, and -4 decimal places, respectively.

  VALUES ( ROUND(873.726, 2),
           ROUND(873.726, 1),
           ROUND(873.726, 0),
           ROUND(873.726,-1),
           ROUND(873.726,-2),
           ROUND(873.726,-3),
           ROUND(873.726,-4) )

  This example returns:

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>873.730</td>
<td>873.700</td>
<td>874.000</td>
<td>870.000</td>
<td>900.000</td>
<td>1000.000</td>
<td>0.000</td>
</tr>
</tbody>
</table>

- Calculate using both positive and negative numbers.

  VALUES ( ROUND(3.5, 0),
           ROUND(3.1, 0),
           ROUND(-3.1, 0),
           ROUND(-3.5, 0) )

  This example returns:

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.0</td>
<td>3.0</td>
<td>-3.0</td>
<td>-4.0</td>
</tr>
</tbody>
</table>

- Calculate the decimal floating-point number 3.12350 rounded to three decimal places.

  VALUES ( ROUND(DECFLOAT('3.12350'), 3))

  This example returns:

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.12400</td>
<td></td>
</tr>
</tbody>
</table>

- Set the host variable RND_DT with the input date rounded to the nearest month value.

  SET :RND_DATE = ROUND(DATE('2000-08-16'), 'MONTH');

  The value set is 2000-09-01.

- Set the host variable RND_TMSTMP with the input timestamp rounded to the nearest year value.

  SET :RND_TMSTMP = ROUND(TIMESTAMP('2000-08-14-17.30.00'), 'YEAR');

  The value set is 2001-01-01-00.00.00.000000.
The schema is SYSIBM.

The ROUND_TIMESTAMP scalar function returns a TIMESTAMP that is the `expression` rounded to the unit specified by the `format-string`. If `format-string` is not specified, `expression` is rounded to the nearest day, as if ‘DD’ is specified for `format-string`.

`expression`
An expression that returns a value of one of the following built-in data types: a DATE or a TIMESTAMP.

`format-string`
An expression that returns a built-in character string data type with an actual length that is not greater than 254 bytes. The format element in `format-string` specifies how `expression` should be rounded. For example, if `format-string` is ‘DD’, the timestamp that is represented by `expression` is rounded to the nearest day. Leading and trailing blanks are removed from the string, and the resulting substring must be a valid format element for a timestamp (SQLSTATE 22007). The default is ‘DD’.

Allowable values for `format-string` are listed in the table of format elements found in the description of the ROUND function.

`locale-name`
A character constant that specifies the locale used to determine the first day of the week when using format model values DAY, DY, or D. The value of `locale-name` is not case sensitive and must be a valid locale (SQLSTATE 42815). For information on valid locales and their naming, see “Locale names for SQL and XQuery”. If `locale-name` is not specified, the value of the special register CURRENT LOCALE LC_TIME is used.

The result of the function is a TIMESTAMP with a timestamp precision of:
- \( p \) when the data type of `expression` is TIMSTAMP\((p)\)
- 0 when the data type of `expression` is DATE
- 6 otherwise.

The result can be null; if any argument is null, the result is the null value.

**Notes**
- **Determinism:** ROUND_TIMESTAMP is a deterministic function. However, the following invocations of the function depend on the value of the special register CURRENT LOCALE LC_TIME.
  - Round of a date or timestamp value when `locale-name` is not explicitly specified and one of the following is true:
    - `format-string` is not a constant
    - `format-string` is a constant and includes format elements that are locale sensitive
  Invocations of the function that depend on the value of a special register cannot be used wherever special registers cannot be used.
Example

- Set the host variable `RND_TMSTMP` with the input timestamp rounded to the nearest year value.

  ```sql
  SET :RND_TMSTMP = ROUND_TIMESTAMP('2000-08-14-17.30.00', 'YEAR');
  ```

  The value set is 2001-01-01-00.00.00.000000.
The schema is SYSIBM.

The RPAD function returns a string composed of string-expression padded on the right, with pad or blanks. The RPAD function treats leading or trailing blanks in string-expression as significant. Padding will only occur if the actual length of string-expression is less than integer, and pad is not an empty string.

string-expression
An expression that specifies the source string. The expression must return a value that is a built-in CHAR, VARCHAR, GRAPHIC, VARGRAPHIC, numeric, or datetime data type. If the value is not a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type, it is implicitly cast to VARCHAR before evaluating the function.

integer
An integer expression that specifies the length of the result. The expression must return a value that is a built-in numeric, CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type. If the value is not of type INTEGER, it is implicitly cast to INTEGER before evaluating the function. The value must be zero or a positive integer that is less than or equal to n, where n is 32 672 if string-expression is a character string, or 16 336 if string-expression is a graphic string.

pad
An expression that specifies the string with which to pad. The expression must return a value that is a built-in CHAR, VARCHAR, GRAPHIC, VARGRAPHIC, numeric, or datetime data type. If the value is not a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type, it is implicitly cast to VARCHAR before evaluating the function.

If pad is not specified, the pad character is determined as follows:
- SBCS blank character if string-expression is a character string.
- Ideographic blank character, if string-expression is a graphic string. For graphic string in an EUC database, X'3000' is used. For graphic string in a Unicode database, X'0020' is used.

The result of the function is a varying length string that has the same code page as string-expression. The value of string-expression and the value of pad must have compatible data types. If the string-expression and pad have different code pages, then pad is converted to the code page of string-expression. If either string-expression or pad is FOR BIT DATA, no character conversion occurs.

The length attribute of the result depends on whether the value for integer is available when the SQL statement containing the function invocation is compiled (for example, if it is specified as a constant or a constant expression) or available only when the function is executed (for example, if it is specified as the result of invoking a function). When the value is available when the SQL statement containing the function invocation is compiled, if integer is greater than zero, the length attribute of the result is integer. If integer is 0, the length attribute of the result is 1. When the value is available only when the function is executed, the length attribute of the result is determined according to the following table:
Table 57. Determining the result length when integer is available only when the function is executed

<table>
<thead>
<tr>
<th>Data type of string-expression</th>
<th>Result data type length</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR(n) or VARCHAR(n)</td>
<td>Minimum of ( n+100 ) and 32 672</td>
</tr>
<tr>
<td>GRAPHIC(n) or VARGRAPHIC(n)</td>
<td>Minimum of ( n+100 ) and 16 336</td>
</tr>
</tbody>
</table>

The actual length of the result is determined from integer. If integer is 0 the actual length is 0, and the result is the empty result string. If integer is less than the actual length of string-expression, the actual length is integer and the result is truncated.

If any argument can be null, the result can be null; if any argument is null, the result is the null value.

**Examples**

- Assume that NAME is a VARCHAR(15) column containing the values “Chris”, “Meg”, and “Jeff”. The following query will completely pad out a value on the right with periods:

  ```sql
  SELECT RPAD(NAME,15,'.') AS NAME FROM T1;
  returns:
  NAME
  --------------
  Chris..........
  Meg............
  Jeff...........
  ```

- Assume that NAME is a VARCHAR(15) column containing the values “Chris”, “Meg”, and “Jeff”. The following query will completely pad out a value on the right with pad (note that in some cases there is a partial instance of the padding specification):

  ```sql
  SELECT RPAD(NAME,15,'123') AS NAME FROM T1;
  returns:
  NAME
  --------------
  Chris1231231231
  Meg123123123123
  Jeff12312312312
  ```

- Assume that NAME is a VARCHAR(15) column containing the values “Chris”, “Meg”, and “Jeff”. The following query will only pad each value to a length of 5:

  ```sql
  SELECT RPAD(NAME,5,'.') AS NAME FROM T1;
  returns:
  NAME
  ----
  Chris
  Meg..
  Jeff.
  ```

- Assume that NAME is a CHAR(15) column containing the values “Chris”, “Meg”, and “Jeff”. Note that the result of RTRIM is a varying length string with the blanks removed:

  ```sql
  SELECT RPAD(RTRIM(NAME),15,'.') AS NAME FROM T1;
  returns:
  ```
• Assume that NAME is a VARCHAR(15) column containing the values “Chris”, “Meg”, and “Jeff”. Note that “Chris” is truncated, “Meg” is padded, and “Jeff” is unchanged:

```sql
SELECT RPAD(NAME, 4, '.') AS NAME FROM T1;
```

returns:

<table>
<thead>
<tr>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chri</td>
</tr>
<tr>
<td>Meg.</td>
</tr>
<tr>
<td>Jeff</td>
</tr>
</tbody>
</table>

**RPAD**

NAME

---------------
Chris..........
Meg............
Jeff...........

Assume that NAME is a VARCHAR(15) column containing the values “Chris”, “Meg”, and “Jeff”. Note that “Chris” is truncated, “Meg” is padded, and “Jeff” is unchanged:

```sql
SELECT RPAD(NAME, 4, '.') AS NAME FROM T1;
```

returns:

<table>
<thead>
<tr>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chri</td>
</tr>
<tr>
<td>Meg.</td>
</tr>
<tr>
<td>Jeff</td>
</tr>
</tbody>
</table>
The schema is SYSIBM. (The SYSFUN version of this function continues to be available with support for CLOB arguments.)

The RTRIM function removes blanks from the end of string-expression.

The expression must return a value that is a built-in CHAR, VARCHAR, GRAPHIC, VARGRAPHIC, numeric, or datetime data type. If the value is not a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type, it is implicitly cast to VARCHAR before evaluating the function.

- If the argument is a graphic string in a DBCS or EUC database, then the trailing double byte blanks are removed.
- If the argument is a graphic string in a Unicode database, then the trailing UCS-2 blanks are removed.
- Otherwise, the trailing single byte blanks are removed.

The result data type of the function is:
- VARCHAR if the data type of string-expression is VARCHAR or CHAR
- VARGRAPHIC if the data type of string-expression is VARGRAPHIC or GRAPHIC

The length parameter of the returned type is the same as the length parameter of the argument data type.

The actual length of the result for character strings is the length of string-expression minus the number of bytes removed for blank characters. The actual length of the result for graphic strings is the length (in number of double byte characters) of string-expression minus the number of double byte blank characters removed. If all of the characters are removed, the result is an empty, varying-length string (length is zero).

If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Example: Assume that host variable HELLO is defined as CHAR(9) and has a value of 'Hello'.

VALUES RTRIM(:HELLO)

The result is 'Hello'.
SECLABEL

The schema is SYSIBM.

The SECLABEL function returns an unnamed security label with a data type of DB2SECURITYLABEL. Use the SECLABEL function to insert a security label with given component values without having to create a named security label.

security-policy-name

A string that specifies a security policy that exists at the current server (SQLSTATE 42704). The string must be a character or graphic string constant or host variable.

security-label-string

An expression that returns a valid representation of a security label for the security policy named by security-policy-name (SQLSTATE 4274I). The expression must return a value that is a built-in CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type.

Examples:

- The following statement inserts a row in table REGIONS which is protected by the security policy named CONTRIBUTIONS. The security label for the row to be inserted is given by the SECLABEL function. The security policy CONTRIBUTIONS has two components. The security label given has the element LIFE MEMBER for first component, the elements BLUE and YELLOW for the second component.

  ```sql
  INSERT INTO REGIONS
  VALUES (SECLABEL('CONTRIBUTIONS', 'LIFE MEMBER:(BLUE,YELLOW)'), 1, 'Northeast')
  ```

- The following statement inserts a row in table CASE_IDS which is protected by the security policy named TS_SECPOLICY, which has three components. The security label is provided by the SECLABEL function. The security label inserted has the element HIGH PROFILE for the first component, the empty value for the second component and the element G19 for the third component.

  ```sql
  INSERT INTO CASE_IDS
  VALUES (SECLABEL('TS_SECPOLICY', 'HIGH PROFILE:():G19') , 3, 'KLB')
  ```
The schema is SYSIBM.

The SECLABEL_BY_NAME function returns the specified security label. The security label returned has a data type of DB2SECURITYLABEL. Use this function to insert a named security label.

security-policy-name
   A string that specifies a security policy that exists at the current server (SQLSTATE 42704). The string must be a character or graphic string constant or host variable.

security-label-name
   An expression that returns the name of a security label that exists at the current server for the security policy named by security-policy-name (SQLSTATE 4274I). The expression must return a value that is a built-in CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type.

Examples:

- User Tina is trying to insert a row in table REGIONS which is protected by the security policy named CONTRIBUTIONS. Tina wants the row to be protected by the security label named EMPLOYEESECLABEL. This statement fails because CONTRIBUTIONS.EMPLOYEESECLABEL is an unknown identifier:

  ```sql
  INSERT INTO REGIONS
  VALUES (CONTRIBUTIONS.EMPLOYEESECLABEL, 1, 'Southwest') -- incorrect
  ```

  This statement fails because the first value is a string, it does not have a data type of DB2SECURITYLABEL:

  ```sql
  INSERT INTO REGIONS
  VALUES ('CONTRIBUTIONS.EMPLOYEESECLABEL', 1, 'Southwest') -- incorrect
  ```

  This statement succeeds because the SECLABEL_BY_NAME function returns a security label that has a data type of DB2SECURITYLABEL:

  ```sql
  INSERT INTO REGIONS
  VALUES (SECLABEL_BY_NAME('CONTRIBUTIONS', 'EMPLOYEESECLABEL'),
          1, 'Southwest') -- correct
  ```
SECLABEL_TO_CHAR

The schema is SYSIBM.

The SECLABEL_TO_CHAR function accepts a security label and returns a string that contains all elements in the security label. The string is in the security label string format.

security-policy-name
   A string that specifies a security policy that exists at the current server (SQLSTATE 42704). The string must be a character or graphic string constant or host variable.

security-label
   An expression that returns a security label value that is valid for the security policy named by security-policy-name (SQLSTATE 4274I). The expression must return a value that is a built-in SYSPROC.DB2SECURITYLABEL distinct type.

Notes
   • If the authorization ID of the statement executes this function on a security label being read from a column with a data type of DB2SECURITYLABEL then that authorization ID’s LBAC credentials might affect the output of the function. In such a case an element is not included in the output if the authorization ID does not have read access to that element. An authorization ID has read access to an element if its LBAC credentials would allow it to read data that was protected by a security label containing only that element, and no others.

   For the rule set DB2LBACRULES only components of type TREE can contain elements that you do not have read access to. For other types of component, if any one of the elements block read access then you will not be able to read the row at all. So only components of type tree will have elements excluded in this way.

Example:
   • The EMP table has two columns, RECORDNUM and LABEL; RECORDNUM has data type INTEGER, and LABEL has type DB2SECURITYLABEL. Table EMP is protected by security policy DATA_ACCESSPOLICY, which uses the DB2LBACRULES rule set and has only one component (GROUPS, of type TREE). GROUPS has five elements: PROJECT, TEST, DEVELOPMENT, CURRENT, AND FIELD. The following diagram shows the relationship of these elements to one another:
The user whose ID is Djavan holds a security label for reading that contains only the DEVELOPMENT element. This means that Djavan has read access to the DEVELOPMENT, CURRENT, and FIELD elements:

```
SELECT RECORDNUM, SECLABEL_TO_CHAR('DATA_ACCESSPOLICY', LABEL) FROM EMP
```

returns:

<table>
<thead>
<tr>
<th>RECORDNUM</th>
<th>LABEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>FIELD</td>
</tr>
<tr>
<td>3</td>
<td>(CURRENT, FIELD)</td>
</tr>
</tbody>
</table>

The row with a RECORDNUM value of 1 is not included in the output, because Djavan’s LBAC credentials do not allow him to read that row. In the row with a RECORDNUM value of 2, element TEST is not included in the output, because Djavan does not have read access to that element; Djavan would not have been able to access the row at all if TEST were the only element in the security label. Because Djavan has read access to elements CURRENT and FIELD, both elements appear in the output.

Now Djavan is granted an exemption to the DB2LBACREADTREE rule. This means that no element of a TREE type component will block read access. The same query returns:

```
SELECT RECORDNUM, LABEL FROM EMP
```

returns:

<table>
<thead>
<tr>
<th>RECORDNUM</th>
<th>LABEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>PROJECT</td>
</tr>
<tr>
<td>2</td>
<td>(TEST, FIELD)</td>
</tr>
<tr>
<td>3</td>
<td>(CURRENT, FIELD)</td>
</tr>
</tbody>
</table>

This time the output includes all rows and all elements, because the exemption gives Djavan read access to all of the elements.
SECOND

The schema is SYSIBM.

The SECOND function returns the seconds part of a value with optional fractional seconds.

**expression**
An expression that returns a value that must be a DATE, TIME, TIMESTAMP, time duration, timestamp duration, or a valid string representation of a date, time, or timestamp that is not a CLOB or DBCLOB.

If **expression** is a DATE or a valid string representation of a date, it is first converted to a TIMESTAMP(0) value, assuming a time of exactly midnight (00.00.00).

Only Unicode databases support an argument that is a graphic string representation of a date, time, or timestamp. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

If **expression** is a character string, leading blanks are included and trailing blanks are removed prior to converting the value to a datetime value. For the valid formats of string representations of datetime values, see “String representations of datetime values” in “Datetime values”.

**integer-constant**
An integer constant representing the scale for the fractional seconds. The value must be in the range 0 through 12.

The result of the function with a single argument is a large integer. The result of the function with two arguments is DECIMAL(2+s,s) where s is the value of the **integer-constant**. If the first argument can be null, the result can be null; if the first argument is null, the result is the null value.

The other rules depend on the data type of the first argument and the number of arguments:

• If the first argument is a DATE, TIME, TIMESTAMP, or valid string representation of a date, time, or timestamp:
  – If only one argument is specified, the result is the seconds part of the value (0 and 59).
  – If both arguments are specified, the result is the seconds part of the value (0 to 59) and **integer-constant** digits of the fractional seconds part of the value where applicable. If there are no fractional seconds in the value, then zeroes are returned.

• If the first argument is a time duration or timestamp duration:
  – If only one argument is specified, the result is the seconds part of the value (-99 to 99).
  – If both arguments are specified, the result is the seconds part of the value (-99 to 99) and **integer-constant** digits of the fractional seconds part of the value where applicable. If there are no fractional seconds in the value then zeroes are returned. A nonzero result has the same sign as the argument.
Examples

- Assume that the host variable TIME_DUR (decimal(6,0)) has the value 153045.
  
  ```sql
  SELECT SECOND(:TIME_DUR)
  FROM SYSIBM.SYSDUMMY1
  ```

  Returns the value 45.

- Assume that the column RECEIVED (whose data type is TIMESTAMP) has an internal value equivalent to 1988-12-25-17.12.30.000000.
  
  ```sql
  SELECT SECOND(RECEIVED)
  FROM IN_TRAY
  ```

  Returns the value 30.

- Get the seconds with fractional seconds from a current timestamp with milliseconds.
  
  ```sql
  SELECT SECOND(CURRENTTIMESTAMP(3), 3)
  FROM SYSIBM.SYSDUMMY1
  ```

  Returns a DECIMAL(5,3) value based on the current timestamp that could be something like 54.321.
SIGN

\[ \text{SIGN}\left(\text{expression}\right) \]

The schema is SYSIBM. (The SYSFUN version of the SIGN function continues to be available.)

Returns an indicator of the sign of the argument. If the argument is less than zero, -1 is returned. If the argument is the decimal floating-point value of -0, the decimal floating-point value of -0 is returned. If argument equals zero, 0 is returned. If argument is greater than zero, 1 is returned.

The argument can be of any built-in numeric data type. DECIMAL and REAL values are converted to double-precision floating-point numbers for processing by the function.

The result of the function is:
- SMALLINT if the argument is SMALLINT
- INTEGER if the argument is INTEGER
- BIGINT if the argument is BIGINT
- DECFLOAT\( (n) \) if the argument is DECFLOAT\( (n) \)
- DOUBLE otherwise.

The result can be null; if the argument is null, the result is the null value.

Example:
- Assume that host variable PROFIT is a large integer with a value of 50000.
  \[ \text{VALUES SIGN\left(:\text{PROFIT}\right)} \]

  Returns the value 1.
The SIN function returns the sine of the argument, where the argument is an angle expressed in radians. The argument can be of any built-in numeric data type (except for DECIMAL). It is converted to double-precision floating-point number for processing by the function.

The result of the function is a double-precision floating-point number. The result can be null if the argument can be null or the database is configured with dft_sqlmathwarn set to YES; the result is the null value if the argument is null.
The schema is SYSIBM.

Returns the hyperbolic sine of the argument, where the argument is an angle expressed in radians.

The argument can be of any built-in numeric data type (except for DECFLOAT). It is converted to a double-precision floating-point number for processing by the function.

The result of the function is a double-precision floating-point number. The result can be null if the argument can be null or the database is configured with `dft_sqlmathwarn` set to YES; the result is the null value if the argument is null.
SMALLINT

Numeric to Smallint:

\[
\text{SMALLINT}\left(-\text{numeric-expression}-\right)
\]

String to Smallint:

\[
\text{SMALLINT}\left(-\text{string-expression}-\right)
\]

The schema is SYSIBM.

The SMALLINT function returns a small integer representation of either:
- A number
- A string representation of a number

**Numeric to Smallint:**

\(\text{numeric-expression}\)

An expression that returns a value of any built-in numeric data type.

The result is the same number that would occur if the argument were assigned to a small integer column or variable. The fractional part of the argument is truncated. If the whole part of the argument is not within the range of small integers, an error is returned (SQLSTATE 22003).

**String to Smallint:**

\(\text{string-expression}\)

An expression that returns a value that is a character-string or Unicode graphic-string representation of a number with a length not greater than the maximum length of a character constant.

The result is the same number that would result from \(\text{CAST(string-expression AS SMALLINT)}\). Leading and trailing blanks are eliminated and the resulting string must conform to the rules for forming an integer, decimal, floating-point, or decimal floating-point constant (SQLSTATE 22018). If the whole part of the argument is not within the range of small integers, an error is returned (SQLSTATE 22003). The data type of \(\text{string-expression}\) must not be CLOB or DBCLOB (SQLSTATE 42884).

The result of the function is a small integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

**Note:** The CAST specification should be used to increase the portability of applications. For more information, see “CAST specification”.

**Example**

Using the EMPLOYEE table, select a list containing salary (SALARY) divided by education level (EDLEVEL). Truncate any decimal in the calculation. The list should also contain the values used in the calculation and the employee number (EMPNO).

\[
\text{SELECT SMALLINT(SALARY / EDLEVEL), SALARY, ESDLEVEL, EMPNO}
\text{ FROM EMPLOYEE}
\]
The schema is SYSFUN.

Returns a 4-character code representing the sound of the words in the argument. The result can be used to compare with the sound of other strings.

The argument can be a character string that is either a CHAR or VARCHAR not exceeding 4000 bytes. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed. The function interprets data that is passed to it as if it were ASCII characters, even if it is encoded in UTF-8.

The result of the function is CHAR(4). The result can be null; if the argument is null, the result is the null value.

The SOUNDEX function is useful for finding strings for which the sound is known but the precise spelling is not. It makes assumptions about the way that letters and combinations of letters sound that can help to search out words with similar sounds. The comparison can be done directly or by passing the strings as arguments to the DIFFERENCE function.

Example:

Using the EMPLOYEE table, find the EMPNO and LASTNAME of the employee with a surname that sounds like 'Loucesy'.

```sql
SELECT EMPNO, LASTNAME FROM EMPLOYEE
WHERE SOUNDEX(LASTNAME) = SOUNDEX('Loucesy')
```

This example returns the following:

```
EMPNO  LASTNAME
-------  ---------------
000110  LUCCHESI
```
The schema is SYSFUN.

Returns a character string consisting of blanks with length specified by the argument.

The argument can be SMALLINT or INTEGER.

The result of the function is VARCHAR(4000). The result can be null; if the argument is null, the result is the null value.
The schema is SYSIBM. (The SYSFUN version of the SQRT function continues to be available.)

The SQRT function returns the square root of a number.

The argument must be an expression that returns a value of any built-in numeric data type. If the argument is decimal floating-point, the operation is performed in decimal floating-point; otherwise, the argument is converted to double-precision floating-point for processing by the function.

If the argument is DECFLOAT(n), the result is DECFLOAT(n); otherwise, the result is a double-precision floating-point number.

The result can be null; if the argument is null, the result is the null value.

Notes
  • Results involving DECFLOAT special values: If the argument is a special decimal floating-point value, the rules for general arithmetic operations for decimal floating-point apply. See “General arithmetic operation rules for decimal floating-point” in “General arithmetic operation rules for decimal floating-point” on page 223.

Examples
  • Assume that SQUARE is a DECIMAL(2,1) host variable with a value of 9.0.
    
    ```sql
    VALUES SQRT(:SQUARE)
    ```

    Returns the approximate value 3.00.
The schema is SYSIBM. The function name cannot be specified as a qualified name when keywords are used in the function signature.

The STRIP function removes blanks or occurrences of another specified character from the end or the beginning of a string expression.

The STRIP function is identical to the TRIM scalar function.

**string-expression**

An expression that specifies the string from which the result is derived. The expression must return a value that is a built-in CHAR, VARCHAR, GRAPHIC, VARGRAPHIC, numeric, or datetime data type. If the value is not a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type, it is implicitly cast to VARCHAR before evaluating the function.

**BOTH, LEADING, or TRAILING**

Specifies whether characters are removed from the beginning, the end, or from both ends of the string expression. If this argument is not specified, the characters are removed from both the end and the beginning of the string.

**strip-character**

A single-character constant that specifies the character that is to be removed. The *strip-character* can be any character whose UTF-32 encoding is a single character or a single digit numeric value. The binary representation of the character is matched.

If *strip-character* is not specified and:

- If the *string-expression* is a DBCS graphic string, the default *strip-character* is a DBCS blank, whose code point is dependent on the database code page
- If the *string-expression* is a UCS-2 graphic string, the default *strip-character* is a UCS-2 blank (X'0020')
- Otherwise, the default *strip-character* is an SBCS blank (X'20')

The result is a varying-length string with the same maximum length as the length attribute of the *string-expression*. The actual length of the result is the length of the *string-expression* minus the number of bytes that are removed. If all of the characters are removed, the result is an empty varying-length string. The code page of the result is the same as the code page of the *string-expression*.

Example:

- Assume that the host variable BALANCE of type CHAR(9) has a value of '000345.50'.
  ```sql
  SELECT STRIP(:BALANCE, LEADING, '0'),
  FROM SYSIBM.SYSDUMMY1
  ```
  returns the value '345.50'.
The SUBSTR function returns a substring of a string.

\textit{string} \\
\hspace{1em} An expression that specifies the string from which the result is derived.

The expression must return a value that is a built-in string, numeric, or datetime data type. If the value is not a string data type, it is implicitly cast to VARCHAR before evaluating the function. If \textit{string} is either a character string or a binary string, a substring of \textit{string} is zero or more contiguous bytes of \textit{string}. If \textit{string} is a graphic string, a substring of \textit{string} is zero or more contiguous double-byte characters of \textit{string}.

\textit{start} \\
\hspace{1em} An expression that specifies the position of the first byte of the result for a character string or a binary string or the position of the first character of the result for a graphic string. The expression must return a value that is a built-in numeric, CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type. If the value is not of type INTEGER, it is implicitly cast to INTEGER before evaluating the function. The integer value must be between 1 and the length or maximum length of \textit{string}, depending on whether \textit{string} is fixed-length or varying-length (SQLSTATE 22011, if out of range). It must be specified as number of bytes in the context of the database code page and not the application code page.

\textit{length} \\
\hspace{1em} An expression that specifies the length of the result. If specified, the expression must return a value that is a built-in numeric, CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type. If the value is not of type INTEGER, it is implicitly cast to INTEGER before evaluating the function. The value of the integer must be in the range of 0 to \( n \), where \( n \) equals (the length attribute of \textit{string}) - \textit{start} + 1 (SQLSTATE 22011, if out of range).

If \textit{length} is explicitly specified, \textit{string} is effectively padded on the right with the necessary number of blank characters (single-byte for character strings; double-byte for graphic strings) or hexadecimal zero characters (for BLOB strings) so that the specified substring of \textit{string} always exists. The default for \textit{length} is the number of bytes from the byte specified by the \textit{start} to the last byte of \textit{string} in the case of character string or binary string or the number of double-byte characters from the character specified by the \textit{start} to the last character of \textit{string} in the case of a graphic string. However, if \textit{string} is a varying-length string with a length less than \textit{start}, the default is zero and the result is the empty string. It must be specified as number of bytes in the context of the database code page and not the application code page. (For example, the column NAME with a data type of VARCHAR(18) and a value of ‘MCKNIGHT’ will yield an empty string with \texttt{SUBSTR(NAME,10)}).

If \textit{string} is a character string, the result of the function is a character string represented in the code page of its first argument. If it is a binary string, the result of the function is a binary string. If it is a graphic string, the result of the function is a graphic string represented in the code page of its first argument.
argument is a host variable that is not a binary string and not a FOR BIT DATA character string, the code page of the result is the database code page. If any argument of the SUBSTR function can be null, the result can be null; if any argument is null, the result is the null value.

Table 58 shows that the result type and length of the SUBSTR function depend on the type and attributes of its inputs.

Table 58. Data Type and Length of SUBSTR Result

<table>
<thead>
<tr>
<th>String Argument Data Type</th>
<th>Length Argument</th>
<th>Result Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR(A)</td>
<td>constant (l&lt;255)</td>
<td>CHAR(l)</td>
</tr>
<tr>
<td>CHAR(A)</td>
<td>not specified but start argument is a constant</td>
<td>CHAR(A-start+1)</td>
</tr>
<tr>
<td>CHAR(A)</td>
<td>not a constant</td>
<td>VARCHAR(A)</td>
</tr>
<tr>
<td>VARCHAR(A)</td>
<td>constant (l&lt;255)</td>
<td>CHAR(l)</td>
</tr>
<tr>
<td>VARCHAR(A)</td>
<td>constant (254&lt;l&lt;32673)</td>
<td>VARCHAR(l)</td>
</tr>
<tr>
<td>VARCHAR(A)</td>
<td>not a constant or not specified</td>
<td>VARCHAR(A)</td>
</tr>
<tr>
<td>CLOB(A)</td>
<td>constant (l)</td>
<td>CLOB(l)</td>
</tr>
<tr>
<td>CLOB(A)</td>
<td>not a constant or not specified</td>
<td>CLOB(A)</td>
</tr>
<tr>
<td>GRAPHIC(A)</td>
<td>constant (l&lt;128)</td>
<td>GRAPHIC(l)</td>
</tr>
<tr>
<td>GRAPHIC(A)</td>
<td>not specified but start argument is a constant</td>
<td>GRAPHIC(A-start+1)</td>
</tr>
<tr>
<td>GRAPHIC(A)</td>
<td>not a constant</td>
<td>VARGRAPHIC(A)</td>
</tr>
<tr>
<td>VARGRAPHIC(A)</td>
<td>constant (l&lt;128)</td>
<td>GRAPHIC(l)</td>
</tr>
<tr>
<td>VARGRAPHIC(A)</td>
<td>constant (127&lt;l&lt;16337)</td>
<td>VARGRAPHIC(l)</td>
</tr>
<tr>
<td>VARGRAPHIC(A)</td>
<td>not a constant</td>
<td>VARGRAPHIC(A)</td>
</tr>
<tr>
<td>DBCLOB(A)</td>
<td>constant (l)</td>
<td>DBCLOB(l)</td>
</tr>
<tr>
<td>DBCLOB(A)</td>
<td>not a constant or not specified</td>
<td>DBCLOB(A)</td>
</tr>
<tr>
<td>BLOB(A)</td>
<td>constant (l)</td>
<td>BLOB(l)</td>
</tr>
<tr>
<td>BLOB(A)</td>
<td>not a constant or not specified</td>
<td>BLOB(A)</td>
</tr>
</tbody>
</table>

**Note:** The LONG VARCHAR and LONG VARGRAPHIC data types continue to be supported but are deprecated and not recommended.

If **string** is a fixed-length string, omission of **length** is an implicit specification of \( \text{LENGTH}(**string**) - \text{start} + 1 \). If **string** is a varying-length string, omission of **length** is an implicit specification of zero or \( \text{LENGTH}(**string**) - \text{start} + 1 \), whichever is greater.

**Notes**

- In dynamic SQL, **string**, **start**, and **length** can be represented by a parameter marker. If a parameter marker is used for **string**, the data type of the operand will be VARCHAR, and the operand will be nullable.

- Though not explicitly stated in the result definitions above, the semantics imply that if **string** is a mixed single- and multi-byte character string, the result might contain fragments of multi-byte characters, depending upon the values of **start** and **length**. For example, the result could possibly begin with the second byte of...
SUBSTR

a multi-byte character, or end with the first byte of a multi-byte character. The
SUBSTR function does not detect such fragments, nor provide any special
processing should they occur.

Examples

• Assume the host variable NAME (VARCHAR(50)) has a value of ‘BLUE JAY’
  and the host variable SURNAME_POS (int) has a value of 6.

  \texttt{SUBSTR(:NAME, :SURNAME\_POS)}

  Returns the value ‘JAY’

  \texttt{SUBSTR(:NAME, :SURNAME\_POS,1)}

  Returns the value ‘J’.

• Select all rows from the PROJECT table for which the project name
  (PROJNAME) starts with the word ‘OPERATION’.

  \texttt{SELECT * FROM PROJECT
  WHERE SUBSTR(PROJNAME,1,10) = ‘OPERATION ’}

  The space at the end of the constant is necessary to preclude initial words such
  as ‘OPERATIONS’.
The schema is SYSIBM.

The SUBSTRB function returns a substring of a string, beginning at a specified position in the string. Lengths are calculated in bytes.

The SUBSTRB function is available starting with version 9.7 Fix Pack 1.

**string**

An expression that specifies the string from which the result is derived.

The expression must return a value that is a built-in string, numeric, or datetime data type. If the value is not a string data type it is implicitly cast to VARCHAR before evaluating the function. A substring of string is zero or more contiguous bytes of string. In a Unicode database, if the value is a graphic data type, it is implicitly cast to a character string data type before evaluating the function.

**start**

An expression that specifies the start position in string of the beginning of the result substring. The expression must return a value that is a built-in numeric, CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type. If the value is not of type INTEGER, it is implicitly cast to INTEGER before evaluating the function.

If start is positive, then the start position is calculated from the beginning of the string. If start is greater than the length of string, then a zero length string is returned.

If start is negative, then the start position is calculated from the end of the string and by counting backwards. If the absolute value of start is greater than the length of string, then a zero length string is returned.

If start is 0, then a start position of 1 is used.

**length**

An expression that specifies the length of the result in bytes. If specified, the expression must return a value that is a built-in numeric, CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type. If the value is not of type INTEGER, it is implicitly cast to INTEGER before evaluating the function.

If the value of length is greater than the number of bytes from the start position to the end of the string, the result length is the length of the first argument minus the start position plus one.

If the value of length is less than or equal to zero, the result of SUBSTRB is a zero length string.

The default for length is the number of bytes from the position specified by start to the last byte of string.

If string is a CHAR or VARCHAR data type, the result of the function is a VARCHAR data type. If it is a CLOB, the result of the function is a CLOB. If it is a BLOB, the result of the function is a BLOB. If the first argument is a host variable
that is not a binary string and not a FOR BIT DATA character string, the code page of the result is the section code page; otherwise, it is the code page of the first argument.

The length attribute of the result is the same as the length attribute of the first argument unless both start and length arguments are specified and defined as constants. In this case, the length attribute of the result is determined as follows:

- If length is a constant which is less than or equal to zero, the length attribute of the result is zero.
- If start is not a constant, but length is a constant, the length attribute of the result is the minimum of the length attribute of the first argument and length.
- If start is a constant, but length is not a constant or not specified, the length attribute of the result is the length attribute of the first argument minus the start position, plus one.
- If start and length are constants, the length attribute of the result is the minimum of the following values:
  - length
  - The length attribute of the first argument minus the start position plus one

If any argument of the SUBSTRB function can be null, the result can be null; if any argument is null, the result is the null value.

Notes

- In dynamic SQL, string, start, and length can be represented by a parameter marker. If a parameter marker is used for string, the data type of the operand will be VARCHAR, and the operand will be nullable.
- Though not explicitly stated in the result definitions above, the semantics imply that if string is a mixed single-byte and multi-byte character string, the result might contain fragments of multi-byte characters, depending on the values of start and length. For example, the result could possibly begin with the second byte of a multi-byte character, or end with the first byte of a multi-byte character. The SUBSTRB function will detect these partial characters and will replace each byte of an incomplete character with a single blank character.
- SUBSTRB is similar to the existing SUBSTR function, with the following exceptions:
  - SUBSTRB supports a negative start value, which indicates the processing should start from the end of the string.
  - SUBSTRB allows length to be greater than the calculated result length. In this case, a shorter string will be returned, rather than returning an error.
  - Graphic input data is not natively supported for the first argument of SUBSTRB. In a Unicode database, graphic data is supported, but it is first converted to character data before evaluating the function, and lengths are calculated in bytes.
  - The result data type of SUBSTRB is VARCHAR if the input data type is CHAR.
  - The length attribute of the result for SUBSTRB is either the same as the length attribute of the first argument, or it is derived based on the start or length attributes, if either of these are constants.

Examples

- Assume the host variable NAME (VARCHAR(50)) has a value of 'BLUE JAY' and the host variable SURNAME_POS (INTEGER) has a value of 6.
SUBSTRB

\text{SUBSTRB}\left(:\text{NAME}, :\text{SURNAME\_POS}\right)

Returns the value 'JAY'.

\text{SUBSTRB}\left(:\text{NAME}, :\text{SURNAME\_POS}, 1\right)

Returns the value 'J'.

- Select all rows from the PROJECT table which end in 'ING'.

\text{SELECT} \ast \text{ FROM} \ \text{PROJECT}
\text{ WHERE} \ \text{SUBSTRB}\left(\text{PROJNAME}, -3\right) = 'ING'
SUBSTRING

SUBSTRING (expression, start, length)

The schema is SYSIBM.

The SUBSTRING function returns a substring of a string.

expression
An expression that specifies the string from which the result is derived. The expression must return a value that is a built-in string, numeric, or datetime data type. If the value is not a string data type, it is implicitly cast to VARCHAR before evaluating the function. If expression is a character string, the result is a character string. If expression is a graphic string, the result is a graphic string. If expression is a binary string, the result is a binary string.

A substring of expression is zero or more contiguous string units of expression.

start
An expression that specifies the position within expression that is to be the first string unit of the result. The expression must return a value that is a built-in numeric, CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type. If the value is not of type INTEGER, it is implicitly cast to INTEGER before evaluating the function. The value of the integer can be positive, negative, or zero; a value of 1 indicates that the first string unit of the result is the first string unit of expression. If OCTETS is specified and expression is graphic data, the value of the integer must be odd; otherwise, an error is returned (SQLSTATE 428GC).

length
The expression must return a value that is a built-in numeric, CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type. If the value is not of type INTEGER, it is implicitly cast to INTEGER before evaluating the function.

If expression is a fixed-length string, omission of length is an implicit specification of CHARACTER_LENGTH(expression USING string-unit) - start + 1, which is the number of string units (CODEUNITS16, CODEUNITS32, or OCTETS) from start to the last position of expression. If expression is a varying-length string, omission of length is an implicit specification of zero or CHARACTER_LENGTH(expression USING string-unit) - start + 1, whichever is greater. If the desired length is zero, the result is the empty string.

If the value is not of type INTEGER, it is implicitly cast to INTEGER before evaluating the function. The value must be greater than or equal to zero. If a value greater than n is specified, where n is the (length attribute of expression) - start + 1, then n is used as the length of the resulting substring. The value is expressed in the units that are explicitly specified. If OCTETS is specified, and expression is graphic data, the value must be an even number (SQLSTATE 428GC).

CODEUNITS16, CODEUNITS32, or OCTETS
Specifies the string unit of start and length. CODEUNITS16 specifies that start
and length are to be expressed in 16-bit UTF-16 code units. CODEUNITS32 specifies that start and length are to be expressed in 32-bit UTF-32 code units. OCTETS specifies that start and length are to be expressed in bytes.

If a string unit is specified as CODEUNITS16 or CODEUNITS32, and expression is a binary string or bit data, an error is returned (SQLSTATE 428GC). If a string unit is specified as OCTETS and expression is a binary string, an error is returned (SQLSTATE 42815).

For more information about CODEUNITS16, CODEUNITS32, and OCTETS, see “String units in built-in functions” in “Character strings”.

When the SUBSTRING function is invoked using OCTETS, and the source-string is encoded in a code page that requires more than one byte per code point (mixed or MBCS), the SUBSTRING operation might split a multi-byte code point and the resulting substring might begin or end with a partial code point. If this occurs, the function replaces the bytes of leading or trailing partial code points with blanks in a way that does not change the byte length of the result. (See a related example below.)

The length attribute of the result is equal to the length attribute of expression. If any argument of the function can be null, the result can be null; if any argument is null, the result is the null value. The result is not padded with any character. If expression has actual length 0, the result also has actual length 0.

Notes:
- The length attribute of the result is equal to the length attribute of the input string expression. This behavior is different from the behavior of the SUBSTR function, where the length attribute is derived from the start and the length arguments of the function.

Examples:
- FIRSTNAME is a VARCHAR(12) column in table T1. One of its values is the 6-character string 'Jürgen'. When FIRSTNAME has this value:

<table>
<thead>
<tr>
<th>Function</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUBSTRING(FIRSTNAME, 1, 2, CODEUNITS32)</td>
<td>'Jü' -- x'4AC3BC'</td>
</tr>
<tr>
<td>SUBSTRING(FIRSTNAME, 1, 2, CODEUNITS16)</td>
<td>'Jü' -- x'4AC3BC'</td>
</tr>
<tr>
<td>SUBSTRING(FIRSTNAME, 1, 2, OCTETS)</td>
<td>'J' -- x'4A20' (a truncated string)</td>
</tr>
<tr>
<td>SUBSTRING(FIRSTNAME, 8, CODEUNITS16)</td>
<td>a zero-length string</td>
</tr>
<tr>
<td>SUBSTRING(FIRSTNAME, 8, 4, OCTETS)</td>
<td>a zero-length string</td>
</tr>
</tbody>
</table>

- The following example illustrates how SUBSTRING replaces the bytes of leading or trailing partial multi-byte code points with blanks when the string length unit is OCTETS. Assume that UTF8_VAR contains the UTF-8 representation of the Unicode string '&N~AB', where '&' is the musical symbol G clef and '~' is the combining tilde character.

```sql
SUBSTRING(UTF8_VAR, 2, 5, OCTETS)
```

Three blank bytes precede the 'N', and one blank byte follows the 'N'.

Chapter 3. Functions  559
The schema is SYSIBM.

The TABLE_NAME function returns an unqualified name of the object found after any alias chains have been resolved. The specified object-name (and object-schema) are used as the starting point of the resolution. If the starting point does not refer to an alias, the unqualified name of the starting point is returned. The resulting name may be of a table, view, or undefined object. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

**object-name**
A character expression representing the unqualified name (usually of an existing alias) to be resolved. object-name must have a data type of CHAR or VARCHAR and a length greater than 0 and less than 129 bytes.

**object-schema**
A character expression representing the schema used to qualify the supplied object-name value before resolution. object-schema must have a data type of CHAR or VARCHAR and a length greater than 0 and less than 129 bytes.

If object-schema is not supplied, the default schema is used for the qualifier.

The data type of the result of the function is VARCHAR(128). If object-name can be null, the result can be null; if object-name is null, the result is the null value. If object-schema is the null value, the default schema name is used. The result is the character string representing an unqualified name. The result name could represent one of the following:

**table**
The value for object-name was either a table name (the input value is returned) or an alias name that resolved to the table whose name is returned.

**view**
The value for object-name was either a view name (the input value is returned) or an alias name that resolved to the view whose name is returned.

**undefined object**
The value for object-name was either an undefined object (the input value is returned) or an alias name that resolved to the undefined object whose name is returned.

Therefore, if a non-null value is given to this function, a value is always returned, even if no object with the result name exists.

**Note:** To improve performance in partitioned database configurations by avoiding the unnecessary communication that occurs between the coordinator partition and catalog partition when using the TABLE_SCHEMA and TABLE_NAME scalar functions, the BASE_TABLE table function can be used instead.
TABLE_SCHEMA

The schema is SYSIBM.

The TABLE_SCHEMA function returns the schema name of the object found after any alias chains have been resolved. The specified object-name (and object-schema) are used as the starting point of the resolution. If the starting point does not refer to an alias, the schema name of the starting point is returned. The resulting schema name may be of a table, view, or undefined object. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

object-name
A character expression representing the unqualified name (usually of an existing alias) to be resolved. object-name must have a data type of CHAR or VARCHAR and a length greater than 0 and less than 129 bytes.

object-schema
A character expression representing the schema used to qualify the supplied object-name value before resolution. object-schema must have a data type of CHAR or VARCHAR and a length greater than 0 and less than 129 bytes.

If object-schema is not supplied, the default schema is used for the qualifier.

The data type of the result of the function is VARCHAR(128). If object-name can be null, the result can be null; if object-name is null, the result is the null value. If object-schema is the null value, the default schema name is used. The result is the character string representing a schema name. The result schema could represent the schema name for one of the following:

table The value for object-name was either a table name (the input or default value of object-schema is returned) or an alias name that resolved to a table for which the schema name is returned.

view The value for object-name was either a view name (the input or default value of object-schema is returned) or an alias name that resolved to a view for which the schema name is returned.

undefined object
The value for object-name was either an undefined object (the input or default value of object-schema is returned) or an alias name that resolved to an undefined object for which the schema name is returned.

Therefore, if a non-null object-name value is given to this function, a value is always returned, even if the object name with the result schema name does not exist. For example, TABLE_SCHEMA('DEPT', 'PEOPLE') returns 'PEOPLE' if the catalog entry is not found.

Note: To improve performance in partitioned database configurations by avoiding the unnecessary communication that occurs between the coordinator partition and catalog partition when using the TABLE_SCHEMA and TABLE_NAME scalar functions, the BASE_TABLE table function can be used instead.

Examples:
PBIRD tries to select the statistics for a given table from SYSCAT.TABLES using an alias PBIRD.A1 defined on the table HEDGES.T1.

```sql
SELECT NPAGES, CARD FROM SYSCAT.TABLES
WHERE TABNAME = TABLE_NAME ('A1')
AND TABSCHEMA = TABLE_SCHEMA ('A1')
```

The requested statistics for HEDGES.T1 are retrieved from the catalog.

- Select the statistics for an object called HEDGES.X1 from SYSCAT.TABLES using HEDGES.X1. Use TABLE_NAME and TABLE_SCHEMA since it is not known whether HEDGES.X1 is an alias or a table.

```sql
SELECT NPAGES, CARD FROM SYSCAT.TABLES
WHERE TABNAME = TABLE_NAME ('X1','HEDGES')
AND TABSCHEMA = TABLE_SCHEMA ('X1','HEDGES')
```

Assuming that HEDGES.X1 is a table, the requested statistics for HEDGES.X1 are retrieved from the catalog.

- Select the statistics for a given table from SYSCAT.TABLES using an alias PBIRD.A2 defined on HEDGES.T2 where HEDGES.T2 does not exist.

```sql
SELECT NPAGES, CARD FROM SYSCAT.TABLES
WHERE TABNAME = TABLE_NAME ('A2','PBIRD')
AND TABSCHEMA = TABLE_SCHEMA ('A2','PBIRD')
```

The statement returns 0 records as no matching entry is found in SYSCAT.TABLES where TABNAME = 'T2' and TABSCHEMA = 'HEDGES'.

- Select the qualified name of each entry in SYSCAT.TABLES along with the final referenced name for any alias entry.

```sql
SELECT TABSCHEMA AS SCHEMA, TABNAME AS NAME,
   TABLE_SCHEMA (BASE_TABNAME, BASE_TABSCHEMA) AS REAL_SCHEMA,
   TABLE_NAME (BASE_TABNAME, BASE_TABSCHEMA) AS REAL_NAME
FROM SYSCAT.TABLES
```

The statement returns the qualified name for each object in the catalog and the final referenced name (after alias has been resolved) for any alias entries. For all non-alias entries, BASE_TABNAME and BASE_TABSCHEMA are null so the REAL_SCHEMA and REAL_NAME columns will contain nulls.
TAN

TAN(expression)

The schema is SYSIBM. (The SYSFUN version of the TAN function continues to be available.)

Returns the tangent of the argument, where the argument is an angle expressed in radians.

The argument can be any built-in numeric data type (except for DECFLOAT). It is converted to a double-precision floating-point number for processing by the function.

The result of the function is a double-precision floating-point number. The result can be null if the argument can be null or the database is configured with `dft_sqlmathwarn` set to YES; the result is the null value if the argument is null.
The schema is SYSIBM.

Returns the hyperbolic tangent of the argument, where the argument is an angle expressed in radians.

The argument can be of any built-in numeric data type (except for DECFLOAT). It is converted to a double-precision floating-point number for processing by the function.

The result of the function is a double-precision floating-point number. The result can be null if the argument can be null or the database is configured with `dft_sqlmathwarn` set to YES; the result is the null value if the argument is null.
The schema is SYSIBM.

The TIME function returns a time from a value.

The argument must be a DATE, TIME, TIMESTAMP, or a valid string representation of a date, time, or timestamp that is not a CLOB or DBCLOB.

Only Unicode databases support an argument that is a graphic string representation of a time or a timestamp. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

The result of the function is a TIME. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The other rules depend on the data type of the argument:

- If the argument is a DATE or string representation of a date:
  - The result is midnight.
- If the argument is a TIME:
  - The result is that time.
- If the argument is a TIMESTAMP:
  - The result is the time part of the timestamp.
- If the argument is a string representation of time or timestamp:
  - The result is the time represented by the string.

**Example**

- Select all notes from the IN_TRAY sample table that were received at least one hour later in the day (any day) than the current time.

  ```sql
  SELECT * FROM IN_TRAY
  WHERE TIME(RECEIVED) >= CURRENT TIME + 1 HOUR
  ```
The schema is SYSIBM.

The `TIMESTAMP` function returns a timestamp from a value or a pair of values.

Only Unicode databases support an argument that is a graphic string representation of a date, a time, or a timestamp. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

The rules for the arguments depend on whether the second argument is specified and the data type of the second argument.

- If only one argument is specified it must be an expression that returns a value of one of the following built-in data types: a `DATE`, a `TIMESTAMP`, or a character string that is not a CLOB. If the argument is a character string, it must be one of the following:
  - A valid character string representation of a date or a timestamp. For the valid formats of string representations of date or timestamp values, see “String representations of datetime values” in “Datetime values”.
  - A character string with an actual length of 13 that is assumed to be a result from the `GENERATE_UNIQUE` function.
  - A string of length 14 that is a string of digits that represents a valid date and time in the form `yyyyxxddhhmmss`, where `yyyy` is the year, `xx` is the month, `dd` is the day, `hh` is the hour, `mm` is the minute, and `ss` is the seconds.

- If both arguments are specified:
  - If the data type of the second argument is not an integer:
    - The first argument must be a `DATE` or a valid string representation of a date and the second argument must be a `TIME` or a valid string representation of a time.
  - If the data type of the second argument is an integer:
    - The first argument must be a `DATE`, `TIMESTAMP`, or a valid string representation of a timestamp or date. The second argument must be an integer constant in the range 0 to 12 representing the timestamp precision.

The result of the function is a `TIMESTAMP`.

The timestamp precision and other rules depend on whether the second argument is specified:

- If both arguments are specified and the second argument is not an integer:
  - The result is a `TIMESTAMP(6)` with the date specified by the first argument and the time specified by the second argument. The fractional seconds part of the timestamp is zero.

- If both arguments are specified and the second argument is an integer:
  - The result is a `TIMESTAMP` with the precision specified in the second argument.

- If only one argument is specified and it is a `TIMESTAMP(p)`:
  - The result is that `TIMESTAMP(p)`.
TIMESTAMP

- If only one argument is specified and it is a DATE:
  - The result is that date with an assumed time of midnight cast to
    TIMESTAMP(0).
- If only one argument is specified and it is a string:
  - The result is the TIMESTAMP(6) value represented by that string extended as
    described earlier with any missing time information. If the argument is a
    string of length 14, the TIMESTAMP has a fractional seconds part of zero.

If the arguments include only date information, the time information in the result
value is all zeros. If either argument can be null, the result can be null; if either
argument is null, the result is the null value.

Examples

- Assume the column START_DATE (whose data type is DATE) has a value
  equivalent to 1988-12-25, and the column START_TIME (whose data type is
  TIME) has a value equivalent to 17.12.30.
  
  `TIMESTAMP(START_DATE, START_TIME)`

  Returns the value '1988-12-25-17.12.30.000000'.

- Convert a timestamp string with 7 digits of fractional seconds to a
  TIMESTAMP(9) value.

  `TIMESTAMP('2007-09-24-15.53.37.2162474', 9)`

  Returns the value '2007-09-24-15.53.37.216247400'.
The schema is SYSIBM.

The `TIMESTAMP_FORMAT` function returns a timestamp that is based on the interpretation of the input string using the specified format.

`string-expression`
- The expression must return a value that is a built-in CHAR or VARCHAR data type. In a Unicode database, if a supplied argument is a GRAPHIC or VARGRAPHIC data type, it is first converted to VARCHAR before evaluating the function. The `string-expression` must contain the components of a timestamp that correspond to the format specified by `format-string`.

`format-string`
- The expression must return a value that is a built-in CHAR or VARCHAR data type. In a Unicode database, if a supplied argument is a GRAPHIC or VARGRAPHIC data type, it is first converted to VARCHAR before evaluating the function. The actual length must not be greater than 254 bytes (SQLSTATE 22007). The value is a template for how `string-expression` is interpreted and then converted to a timestamp value.

A valid `format-string` must contain at least one format element, must not contain multiple specifications for any component of a timestamp, and can contain any combination of the format elements, unless otherwise noted in Table 59 on page 569 (SQLSTATE 22007). For example, `format-string` cannot contain both YY and YYYY, because they are both used to interpret the year component of `string-expression`. Refer to the table to determine which format elements cannot be specified together. Two format elements can optionally be separated by one or more of the following separator characters:

- minus sign (-)
- period (.)
- slash (/)
- comma (,)
- apostrophe (')
- semi-colon (;)
- colon (:)
- blank ( )

Separator characters can also be specified at the start or end of `format-string`. These separator characters can be used in any combination in the format string, for example 'YYYY/MM-DD HH:MM:SS'.
### Table 59. Format elements for the TIMESTAMP_FORMAT function

<table>
<thead>
<tr>
<th>Format element</th>
<th>Related components of a timestamp</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AM or PM</td>
<td>hour</td>
<td>Meridian indicator (morning or evening) without periods. This format element is dependent on locale-name, if specified. Otherwise, it is dependent on the value of the special register CURRENT LOCALE LC_TIME.</td>
</tr>
<tr>
<td>A.M. or P.M.</td>
<td>hour</td>
<td>Meridian indicator (morning or evening) with periods. This format element uses the exact strings “A.M.” or “P.M.” and is independent of the locale name in effect.</td>
</tr>
<tr>
<td>DAY, Day, or day</td>
<td>none</td>
<td>Name of the day in uppercase, titlecase, or lowercase format. The language used is dependent on locale-name, if specified; otherwise, it is dependent on the value of the special register CURRENT LOCALE LC_TIME.</td>
</tr>
<tr>
<td>DY, Dy, or dy</td>
<td>none</td>
<td>Abbreviated name of the day in uppercase, titlecase, or lowercase format. The language used is dependent on locale-name, if specified; otherwise, it is dependent on the value of the special register CURRENT LOCALE LC_TIME.</td>
</tr>
<tr>
<td>D</td>
<td>none</td>
<td>Day of the week (1-7). The first day of the week is dependent on locale-name, if specified; otherwise, it is dependent on the value of the special register CURRENT LOCALE LC_TIME.</td>
</tr>
<tr>
<td>DD</td>
<td>day</td>
<td>Day of month (01-31).</td>
</tr>
<tr>
<td>DDD</td>
<td>month, day</td>
<td>Day of year (001-366).</td>
</tr>
</tbody>
</table>
### Table 59. Format elements for the `TIMESTAMP_FORMAT` function (continued)

<table>
<thead>
<tr>
<th>Format element</th>
<th>Related components of a timestamp</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FF or FF(n)</td>
<td>fractional seconds</td>
<td>Fractional seconds ((0-999999999999)). The number (n) is used to specify the number of digits expected in the string-expression. Valid values for (n) are 1-12 with no leading zeros. Specifying FF is equivalent to specifying FF6. When FF is the last format element and the number of digits for the fractional seconds is less than what is specified by the format, zero digits are padded onto the right of the specified digits.</td>
</tr>
<tr>
<td>HH</td>
<td>hour</td>
<td>HH behaves the same as HH12.</td>
</tr>
<tr>
<td>HH12</td>
<td>hour</td>
<td>Hour of the day (01-12) in 12-hour format. AM is the default meridian indicator.</td>
</tr>
<tr>
<td>HH24</td>
<td>hour</td>
<td>Hour of the day (00-24) in 24-hour format.</td>
</tr>
<tr>
<td>J</td>
<td>year, month, and day</td>
<td>Julian day (number of days since January 1, 4713 BC).</td>
</tr>
<tr>
<td>MI</td>
<td>minute</td>
<td>Minute (00-59).</td>
</tr>
<tr>
<td>MM</td>
<td>month</td>
<td>Month (01-12).</td>
</tr>
<tr>
<td>MONTH, Month, or month</td>
<td>month</td>
<td>Name of the month in uppercase, titlecase, or lowercase format. The language used is dependent on locale-name, if specified; otherwise, it is dependent on the value of the special register CURRENT LOCALE LC_TIME.</td>
</tr>
<tr>
<td>MON, Mon, or mon</td>
<td>month</td>
<td>Abbreviated name of the month in uppercase, titlecase, or lowercase format. The language used is dependent on locale-name, if specified; otherwise, it is dependent on the value of the special register CURRENT LOCALE LC_TIME.</td>
</tr>
<tr>
<td>NNNNNN</td>
<td>microseconds</td>
<td>Microseconds ((000000-999999)). Same as FF6.</td>
</tr>
<tr>
<td>RR</td>
<td>year</td>
<td>Last two digits of the adjusted year (00-99).</td>
</tr>
</tbody>
</table>
Table 59. Format elements for the `TIMESTAMP_FORMAT` function (continued)

<table>
<thead>
<tr>
<th>Format element</th>
<th>Related components of a timestamp</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RRRR</td>
<td>year</td>
<td>4-digit adjusted year (0000-9999).</td>
</tr>
<tr>
<td>SS</td>
<td>seconds</td>
<td>Seconds (00-59).</td>
</tr>
<tr>
<td>SSSSS</td>
<td>hours, minutes, and seconds</td>
<td>Seconds since previous midnight (00000-86400).</td>
</tr>
<tr>
<td>Y</td>
<td>year</td>
<td>Last digit of the year (0-9). First three digits of the current year are used to determine the full 4-digit year.</td>
</tr>
<tr>
<td>YY</td>
<td>year</td>
<td>Last two digits of the year (00-99). First two digits of the current year are used to determine the full 4-digit year.</td>
</tr>
<tr>
<td>YYY</td>
<td>year</td>
<td>Last three digits of the year (000-999). First digit of the current year is used to determine the full 4-digit year.</td>
</tr>
<tr>
<td>YYYY</td>
<td>year</td>
<td>4-digit year (0000-9999).</td>
</tr>
</tbody>
</table>

**Note:** The format elements in Table 59 on page 569 are not case sensitive, except for the following:
- AM, PM
- A.M., P.M.
- DAY, Day, day
- DY, Dy, dy
- D
- MONTH, Month, month
- MON, Mon, mon

The DAY, Day, day, DY, Dy, dy, and D format elements do not contribute to any components of the resulting timestamp. However, a specified value for any of these format elements must be correct for the combination of the year, month, and day components of the resulting timestamp (SQLSTATE 22007). For example, assuming a value of ‘en_US’ is used for `locale-name`, a value of ‘Monday 2008-10-06’ for `string-expression` is valid for a value of ‘Day YYYY-MM-DD’. However, value of ‘Tuesday 2008-10-06’ for `string-expression` would result in error for the same `format-string`.

The RR and RRRR format elements can be used to alter how a specification for a year is to be interpreted by adjusting the value to produce a 2-digit value or a 4-digit value depending on the leftmost two digits of the current year according to the following table.
For example, if the current year is 2007, '86' with format 'RR' means 1986, but if the current year is 2052, it means 2086.

The following defaults are used when a format-string does not include a format element for one of the following components of a timestamp:

<table>
<thead>
<tr>
<th>Timestamp component</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>year</td>
<td>current year, as 4 digits</td>
</tr>
<tr>
<td>month</td>
<td>current month, as 2 digits</td>
</tr>
<tr>
<td>day</td>
<td>01 (first day of the month)</td>
</tr>
<tr>
<td>hour</td>
<td>00</td>
</tr>
<tr>
<td>minute</td>
<td>00</td>
</tr>
<tr>
<td>second</td>
<td>00</td>
</tr>
<tr>
<td>fractional seconds</td>
<td>a number of zeros matching the timestamp precision of the result</td>
</tr>
</tbody>
</table>

Leading zeros can be specified for any component of the timestamp value (that is, month, day, hour, minutes, seconds) that does not have the maximum number of significant digits for the corresponding format element in the format-string.

A substring of the string-expression representing a component of a timestamp (such as year, month, day, hour, minutes, seconds) can include less than the maximum number of digits for that component of the timestamp indicated by the corresponding format element. Any missing digits default to zero. For example, with a format-string of 'YYYY-MM-DD HH24:MI:SS', an input value of '999-3-9 5:7:2' would produce the same result as '0999-03-09 05:07:02'.

If format-string is not specified, string-expression will be interpreted using a default format based on the value of the special register CURRENT_LOCALE LC_TIME.

precision-constant

An integer constant that specifies the timestamp precision of the result. The value must be in the range 0 to 12. If not specified, the timestamp precision defaults to 6.

locale-name

A character constant that specifies the locale used for the following format elements:

- AM, PM
- DAY, Day, day
- DY, Dy, dy
- D
TIMESTAMP_FORMAT

- MONTH, Month, month
- MON, Mon, mon

The value of locale-name is not case sensitive and must be a valid locale (SQLSTATE 42815). For information on valid locales and their naming, see “Locale names for SQL and XQuery” in the Globalization Guide. If locale-name is not specified, the value of the special register CURRENT LOCALE LC_TIME is used.

The result of the function is a TIMESTAMP with a precision based on precision-constant. If either of the first two arguments can be null, the result can be null; if either of the first two arguments is null, the result is the null value.

Notes
- Julian and Gregorian calendar: The transition from the Julian calendar to the Gregorian calendar on 15 October 1582 is taken into account by this function.
- Determinism: TIMESTAMP_FORMAT is a deterministic function. However, the following invocations of the function depend on the value of either the special register CURRENT LOCALE LC_TIME or CURRENT TIMESTAMP:
  - When format-string is not explicitly specified, or when locale-name is not explicitly specified and one of the following is true:
    - format-string is not a constant
    - format-string is a constant and includes format elements that are locale sensitive
    - format-string is a constant and does not include a format element that fully defines the year (that is, J or YYYY) and so uses the value of the current year
    - format-string is a constant and does not include a format element that fully defines the month (for example, J, MM, MONTH, or MON) and so uses the value of the current month

These invocations that depend on the value of a special register cannot be used wherever special registers cannot be used (SQLSTATE 42621 or 428EC).
- Syntax alternatives: TO_DATE and TO_TIMESTAMP are synonyms for TIMESTAMP_FORMAT.

Examples
- Insert a row into the IN_TRAY table with a receiving timestamp that is equal to one second before the beginning of the year 2000 (December 31, 1999 at 23:59:59).
  
  ```sql
  INSERT INTO IN_TRAY (RECEIVED)
  ```

- An application receives strings of date information into a variable called INDATEVAR. This value is not strictly formatted and might include two or four digits for years, and one or two digits for months and days. Date components might be separated with minus sign (-) or slash (/) characters and are expected to be in day, month, and year order. Time information consists of hours (in 24-hour format) and minutes, and is usually separated by a colon. Sample values include '15/12/98 13:48' and '9-3-2004 8:02'. Insert such values into the IN_TRAY table.

  ```sql
  INSERT INTO IN_TRAY (RECEIVED)
  VALUES (TIMESTAMP_FORMAT(:INDATEVAR, 'DD/MM/RRRR HH24:MI'))
  ```
The use of RRRR in the format allows for 2- and 4-digit year values and assigns missing first two digits based on the current year. If YYYY were used, input values with a 2-digit year would have leading zeros. The slash separator also allows the minus sign character. Assuming a current year of 2007, resulting timestamps from the sample values are:

'9-3-2004 8:02'   --> 2004-03-09-08.02.00.000000
The schema is SYSFUN.

Returns a timestamp value based on a date, time, or timestamp argument. If the argument is a DATE, it inserts zero for all the time elements. If the argument is a TIME, it inserts the value of the CURRENT DATE special register for the date elements, and zero for the fractional time element.

The expression must return a value that is a built-in CHAR, VARCHAR, DATE, TIME, or TIMESTAMP data type. In a Unicode database, if a supplied argument has a GRAPHIC or VARGRAPHIC data type, it is first converted to a character string before evaluating the function. A string expression must return a valid character string representation of a date or timestamp.

The TIMESTAMP_ISO function is generally defined as deterministic. If the first argument has the TIME data type, then the function is not deterministic because the CURRENT DATE is used for the date portion of the timestamp value.

The result of the function is a TIMESTAMP. The result can be null; if the argument is null, the result is the null value.
TIMESTAMPDIFF

TIMESTAMPDIFF

TIMESTAMPDIFF

➤➤TIMESTAMPDIFF(—expression—,—expression—)➤➤

The schema is SYSFUN.

Returns an estimated number of intervals of the type defined by the first argument, based on the difference between two timestamps.

The first argument can be either INTEGER or SMALLINT. Valid values of interval (the first argument) are:

1 Fractions of a second
2 Seconds
4 Minutes
8 Hours
16 Days
32 Weeks
64 Months
128 Quarters
256 Years

The second argument is the result of subtracting two timestamps and converting the result to CHAR(22). The string value must not have more than 6 digits to the right of a decimal point. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

The result of the function is INTEGER. The result can be null; if the argument is null, the result is the null value.

The following assumptions may be used in estimating a difference:
• There are 365 days in a year.
• There are 30 days in a month.
• There are 24 hours in a day.
• There are 60 minutes in an hour.
• There are 60 seconds in a minute.

These assumptions are used when converting the information in the second argument, which is a timestamp duration, to the interval type specified in the first argument. The returned estimate may vary by a number of days. For example, if the number of days (interval 16) is requested for the difference between '1997-03-01-00.00.00' and '1997-02-01-00.00.00', the result is 30. This is because the difference between the timestamps is 1 month, and the assumption of 30 days in a month applies.

Example:

The following example returns 4277, the number of minutes between two timestamps:
TIMESTAMPDIFF

TIMESTAMPDIFF(4, CHAR(TIMESTAMP('2001-09-29-11.25.42.483219') -
TIMESTAMP('2001-09-26-12.07.58.065497')))
TO_CHAR

Character to varchar

```
TO_CHAR(character-expression)
```

Timestamp to varchar:

```
TO_CHAR(timestamp-expression, format-string, locale-name)
```

Decimal floating-point to varchar:

```
TO_CHAR(decimal-floating-point-expression, format-string)
```

The schema is SYSIBM.

The TO_CHAR function returns a character representation of an input expression that has been formatted using a character template.

The TO_CHAR scalar function is a synonym for the VARCHAR_FORMAT scalar function.
The schema is SYSIBM.
The TO_CLOB function returns a CLOB representation of a character string type.
The TO_CLOB scalar function is a synonym for the CLOB scalar function.
The schema is SYSIBM.

The TO_DATE function returns a timestamp that is based on the interpretation of the input string using the specified format.

The TO_DATE scalar function is a synonym for the TIMESTAMP_FORMAT scalar function.
TO_NUMBER

The schema is SYSIBM.

The TO_NUMBER function returns a DECFLOAT(34) value that is based on the interpretation of the input string using the specified format.

The TO_NUMBER scalar function is a synonym for the DECFLOAT_FORMAT scalar function.
The schema is SYSIBM.

The TO_TIMESTAMP function returns a timestamp that is based on the interpretation of the input string using the specified format.

The TO_TIMESTAMP scalar function is a synonym for the TIMESTAMP_FORMAT scalar function except that the default value for \textit{precision-constant} is 12.
The schema is SYSIBM.

The TOTALORDER function returns a SMALLINT value of -1, 0, or 1 that indicates the comparison order of two arguments.

\textit{decfloat-expression1}

An expression that returns a value of any built-in numeric data type. If the argument is not DECFLOAT(34), it is logically converted to DECFLOAT(34) for processing.

\textit{decfloat-expression2}

An expression that returns a value of any built-in numeric data type. If the argument is not a decimal floating-point value, it is converted to DECFLOAT(34) for processing.

Numeric comparison is exact, and the result is determined for finite operands as if range and precision were unlimited. An overflow or underflow condition cannot occur.

If one value is DECFLOAT(16) and the other is DECFLOAT(34), the DECFLOAT(16) value is converted to DECFLOAT(34) before the comparison is made.

The semantics of the TOTALORDER function are based on the total order predicate rules of IEEE 754R. TOTALORDER returns the following values:

\begin{itemize}
  \item -1 if \textit{decfloat-expression1} is lower in order compared to \textit{decfloat-expression2}
  \item 0 if both \textit{decfloat-expression1} and \textit{decfloat-expression2} have the same order
  \item 1 if \textit{decfloat-expression1} is higher in order compared to \textit{decfloat-expression2}
\end{itemize}

The ordering of the special values and finite numbers is as follows:

\texttt{-NAN<-SNAN<-INFINITY<-0.10<-0.100<-0.00<-0.000<0.00<0.10<0.100<INFINITY<SNAN<NAN}

The result of the function is a SMALLINT value. If either argument can be null, the result can be null; if either argument is null, the result is the null value.

Examples:

\begin{itemize}
  \item The following examples show the use of the TOTALORDER function to compare decimal floating point values:
    \begin{verbatim}
    TOTALORDER(-INFINITY, -INFINITY) = 0
    TOTALORDER(DECFLOAT(-1.0), DECFLOAT(-1.0)) = 0
    TOTALORDER(DECFLOAT(-1.0), DECFLOAT(-1.00)) = -1
    TOTALORDER(DECFLOAT(-1.0), DECFLOAT(-0.5)) = -1
    TOTALORDER(DECFLOAT(-1.0), DECFLOAT(0.5)) = -1
    TOTALORDER(DECFLOAT(-1.0), INFINITY) = -1
    TOTALORDER(DECFLOAT(-1.0), SNAN) = -1
    TOTALORDER(DECFLOAT(-1.0), NAN) = -1
    TOTALORDER(NAN, DECFLOAT(-1.0)) = 1
    TOTALORDER(-NAN, -NAN) = 0
    TOTALORDER(-SNAN, -SNAN) = 0
    TOTALORDER(NAN, NAN) = 0
    TOTALORDER(SNAN, SNAN) = 0
    TOTALORDER(-1.0, -1.0) = 0
    TOTALORDER(-1.0, -1.00) = -1
    \end{verbatim}
\end{itemize}
TOTALORDER

TOTALORDER(-1.0, -0.5) = -1
TOTALORDER(-1.0, 0.5) = -1
TOTALORDER(-1.0, INFINITY) = -1
TOTALORDER(-1.0, SNAN) = -1
TOTALORDER(-1.0, NAN) = -1
The TRANSLATE function returns a value in which one or more characters in a string expression might have been converted to other characters.

The function converts all the characters in char-string-exp or graphic-string-exp that also occur in from-string-exp to the corresponding characters in to-string-exp or, if no corresponding characters exist, to the pad character specified by pad-char-exp.

char-string-exp or graphic-string-exp
Specifies a string that is to be converted. The expression must return a value that is a built-in CHAR, VARCHAR, GRAPHIC, VARGRAPHIC, numeric, or datetime data type. If the value is not a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type, it is implicitly cast to VARCHAR before evaluating the function.

to-string-exp
Specifies a string of characters to which certain characters in char-string-exp will be converted.

The expression must return a value that is a built-in CHAR, VARCHAR, GRAPHIC, VARGRAPHIC, numeric, or datetime data type. If the value is not a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type, it is implicitly cast to VARCHAR before evaluating the function. If a value for to-string-exp is not specified, and the data type is not graphic, all characters in char-string-exp will be in monocase; that is, the characters a-z will be converted to the characters A-Z, and other characters will be converted to their uppercase equivalents, if they exist. For example, in code page 850, é maps to É, but ý is not mapped, because code page 850 does not include Ý. If the code point length of the result character is not the same as the code point length of the source character, the source character is not converted.

from-string-exp
Specifies a string of characters which, if found in char-string-exp, will be converted to the corresponding character in to-string-exp.

The expression must return a value that is a built-in CHAR, VARCHAR, GRAPHIC, VARGRAPHIC, numeric, or datetime data type. If the value is not
a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type, it is implicitly cast to VARCHAR before evaluating the function. If \textit{from-string-exp} contains duplicate characters, the first one found will be used, and the duplicates will be ignored. If \textit{to-string-exp} is longer than \textit{from-string-exp}, the surplus characters will be ignored. If \textit{to-string-exp} is specified, \textit{from-string-exp} must also be specified.

\textit{pad-char-exp}

Specifies a single character that will be used to pad \textit{to-string-exp} if \textit{to-string-exp} is shorter than \textit{from-string-exp}. The expression must return a value that is a built-in CHAR, VARCHAR, GRAPHIC, VARGRAPHIC, numeric, or datetime data type. If the value is not a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type, it is implicitly cast to VARCHAR before evaluating the function. The value must have a length attribute of one. If a value is not specified a single-byte blank character is assumed.

With \textit{graphic-string-exp}, only \textit{pad-char-exp} is optional (if a value is not specified, the double-byte blank character is assumed), and each argument, including the pad character, must be of a graphic data type.

The data type and code page of the result is the same as the data type and code page of the first argument. If the first argument is a host variable, the code page of the result is the database code page. Each argument other than the first argument is converted to the result code page unless it or the first argument is defined as FOR BIT DATA, in which case no conversion is done.

In a Unicode database where character and graphic are considered to be equivalent data types, there are the following exceptions:

- The code page of the result is 1208 if any argument but the first argument is FOR BIT DATA.
- The code page of the result is the code page that appears most often in the set of arguments, when no argument is FOR BIT DATA.
- The code page of the result is 1200 when two different code pages appear equally often in the set of arguments, when no argument is FOR BIT DATA.

The length attribute of the result is the same as that of the first argument. If any argument can be null, the result can be null. If any argument is null, the result is the null value.

If the arguments are of data type CHAR or VARCHAR, the corresponding characters in \textit{to-string-exp} and \textit{from-string-exp} must have the same number of bytes. For example, it is not valid to convert a single-byte character to a multi-byte character, or to convert a multi-byte character to a single-byte character. The \textit{pad-char-exp} argument cannot be the first byte of a valid multi-byte character (SQLSTATE 42815).

The characters are matched using a binary comparison. The database collation is not used.

If only \textit{char-string-exp} is specified, single-byte characters will be monocased, and multi-byte characters will remain unchanged.

Examples:

- Assume that the host variable SITE (VARCHAR(30)) has a value of ‘Hanauma Bay’.
\texttt{TRANSLATE(\texttt{:SITE})}

Returns the value ‘HANAUMA BAY’.

\texttt{TRANSLATE(\texttt{:SITE, 'j', 'B'})}

Returns the value ‘Hanauma jay’.

\texttt{TRANSLATE(\texttt{:SITE, 'ei', 'aa'})}

Returns the value ‘Heneume Bey’.

\texttt{TRANSLATE(\texttt{:SITE, 'bA', 'Bay', '%'})}

Returns the value ‘HAnAumA bA%’.

\texttt{TRANSLATE(\texttt{:SITE, 'r', 'Bu'})}

Returns the value ‘Hana ma ray’.
The TRIM function removes blanks or occurrences of another specified character from the end or the beginning of a string expression.

**BOTH, LEADING, or TRAILING**
Specifies whether characters are removed from the beginning, the end, or from both ends of the string expression. If this argument is not specified, the characters are removed from both the end and the beginning of the string.

**strip-character**
A single-character constant that specifies the character that is to be removed. The `strip-character` can be any character whose UTF-32 encoding is a single character or a single digit numeric value. The binary representation of the character is matched.

If `strip-character` is not specified and:
- If the `string-expression` is a DBCS graphic string, the default `strip-character` is a DBCS blank, whose code point is dependent on the database code page
- If the `string-expression` is a UCS-2 graphic string, the default `strip-character` is a UCS-2 blank (X'0020')
- Otherwise, the default `strip-character` is an SBCS blank (X'20')

**FROM string-expression**
The expression must return a value that is a built-in CHAR, VARCHAR, GRAPHIC, VARGRAPHIC, numeric, or datetime data type. If the value is not a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type, it is implicitly cast to VARCHAR before evaluating the function.

The result is a varying-length string with the same maximum length as the length attribute of the `string-expression`. The actual length of the result is the length of the `string-expression` minus the number of bytes that are removed. If all of the characters are removed, the result is an empty varying-length string. The code page of the result is the same as the code page of the `string-expression`.

Examples:
- Assume that the host variable `HELLO` of type CHAR(9) has a value of ' Hello'.
  ```sql
  SELECT TRIM(:HELLO),
          TRIM(LEADING FROM :HELLO)
  FROM SYSIBM.SYSDUMMY1
  ```
  returns the values 'Hello' and ' Hello', respectively.
- Assume that the host variable `BALANCE` of type CHAR(9) has a value of '000345.50'.
  ```sql
  ```
SELECT TRIM(L '0' FROM :BALANCE),
FROM SYSIBM.SYSDUMMY1

returns the value '345.50'.
TRIM_ARRAY

TRIM_ARRAY

```
TRIM_ARRAY
  (array-variable, numeric-expression)
```

The schema is SYSIBM.

The TRIM_ARRAY function deletes elements from the end of an array.

*array-variable*

An SQL variable, SQL parameter, or global variable of an ordinary array type, or a CAST specification of a parameter marker to an ordinary array type. An associative array data type cannot be specified (SQLSTATE 42884).

*numeric-expression*

Specifies the number of elements trimmed from the end of the array. The *numeric-expression* can be of any numeric data type with a value that can be cast to INTEGER. The value of *numeric-expression* must be between 0 and the cardinality of *array-variable* (SQLSTATE 2202E). The default is 1.

The function returns a value with the same array type as *array-variable* but with the cardinality reduced by the value of INTEGER(*numeric-expression*). The result can be null; if either argument is null, the result is the null value.

**Rules**

The TRIM_ARRAY function is not supported for associative arrays (SQLSTATE 42884).

The TRIM_ARRAY function can only be used on the right side of an assignment statement in contexts where arrays are supported (SQLSTATE 42884).

**Examples**

- Remove the last element from the array variable RECENT_CALLS.
  ```
  SET RECENT_CALLS = TRIM_ARRAY(RECENT_CALLS, 1)
  ```

- Assign only the first two elements from the array variable SPECIALNUMBERS to the SQL array variable EULER_CONST:
  ```
  SET EULER_CONST = TRIM_ARRAY(SPECIALNUMBERS, 8)
  ```

  The result is that EULER_CONST will be assigned an array with two elements, the first element value is 2.71828183 and the second element value is the null value.
The schema is SYSIBM.

The TRUNC_TIMESTAMP scalar function returns a TIMESTAMP that is the expression truncated to the unit specified by the format-string. If format-string is not specified, expression is truncated to the nearest day, as if ‘DD’ was specified for format-string.

expression
An expression that returns a value of one of the following built-in data types: a DATE or a TIMESTAMP.

format-string
An expression that returns a built-in character string data type with an actual length that is not greater than 254 bytes. The format element in format-string specifies how expression should be truncated. For example, if format-string is ‘DD’, a timestamp that is represented by expression is truncated to the nearest day. Leading and trailing blanks are removed from the string, and the resulting substring must be a valid format element for a timestamp (SQLSTATE 22007). The default is ‘DD’.

Allowable values for format-string are listed in the table of format elements found in the description of the ROUND function.

locale-name
A character constant that specifies the locale used to determine the first day of the week when using format model values DAY, DY, or D. The value of locale-name is not case sensitive and must be a valid locale (SQLSTATE 42815). For information on valid locales and their naming, see “Locale names for SQL and XQuery”. If locale-name is not specified, the value of the special register CURRENT LOCALE LC_TIME is used.

The result of the function is a TIMESTAMP with the same timestamp precision as expression. The result can be null; if any argument is null, the result is the null value.

The result of the function is a TIMESTAMP with a timestamp precision of:
• $p$ when the data type of expression is TIMESTAMP($p$)
• 0 when the data type of expression is DATE
• 6 otherwise

The result can be null; if any argument is null, the result is the null value.

Notes
• Determinism: TRUNC_TIMESTAMP is a deterministic function. However, the following invocations of the function depend on the value of the special register CURRENT LOCALE LC_TIME.
  - Truncate of a date or timestamp value when locale-name is not explicitly specified and one of the following is true:
    - format-string is not a constant
TRUNC_TIMESTAMP

- format-string is a constant and includes format elements that are locale sensitive.

Invocations of the function that depend on the value of a special register cannot be used wherever special registers cannot be used.

Examples

- Set the host variable TRNK_TMSTMP with the current year rounded to the nearest year value.

  ```sql
  SET :TRNK_TMSTMP = TRUNC_TIMESTAMP('2000-03-14-17.30.00', 'YEAR');
  ```

  The host variable TRNK_TMSTMP is set with the value 2000-01-01-00.00.00.000000.
TRUNCATE or TRUNC

TRUNCATE numeric:

\[
\text{TRUNCATE} \left( \text{numeric-expression1} \right) \text{, numeric-expression2} \]

TRUNCATE datetime:

\[
\text{TRUNCATE} \left( \text{datetime-expression} \right) \text{, 'DD' format-string locale-name} \]

The schema is SYSIBM. The SYSFUN version of the TRUNCATE numeric function continues to be available.

The TRUNCATE function returns a truncated value of:

- A number, truncated to the specified number of places to the right or left of the decimal point, if the result of the first argument is a numeric value
- A datetime value, truncated to the unit specified by format-string, if the first argument is a DATE, TIME, or TIMESTAMP

TRUNCATE numeric

If numeric-expression1 has a numeric data type, the TRUNCATE function returns numeric-expression1 truncated to numeric-expression2 places to the right of the decimal point if numeric-expression2 is positive, or to the left of the decimal point if numeric-expression2 is zero or negative. If numeric-expression2 is not specified, numeric-expression1 is truncated to zero places left of the decimal point.

numeric-expression1

An expression that must return a value that is a built-in CHAR, VARCHAR, GRAPHIC, VARGRAPHIC, or numeric data type. If the value is not a numeric data type, it is implicitly cast to DECFLOAT(34) before evaluating the function.

If the expression is a decimal floating-point data type, the DECFLOAT rounding mode will not be used. The rounding behavior of TRUNCATE corresponds to a value of ROUND_DOWN. If a different rounding behavior is wanted, use the QUANTIZE function.

numeric-expression2

An expression that returns a value that is a built-in numeric data type. If the value is not of type INTEGER, it is implicitly cast to INTEGER before evaluating the function.

If numeric-expression2 is not negative, numeric-expression1 is truncated to the absolute value of numeric-expression2 number of places to the right of the decimal point.

If numeric-expression2 is negative, numeric-expression1 is truncated to the absolute value of numeric-expression2 + 1 number of places to the left of the decimal point. If the absolute value of a negative numeric-expression2 is larger than the number of digits to the left of the decimal point, the result is 0. For example:

\[
\text{TRUNCATE}(748.58, -4) = 0
\]
TRUNCATE or TRUNC

The data type, length, and scale attributes of the result are the same as the data type, length, and scale attributes of the first argument.

If either argument can be null, the result can be null. If either argument is null, the result is the null value.

TRUNCATE datetime

If datetime-expression has a datetime data type, the TRUNCATE function returns datetime-expression rounded to the unit specified by the format-string. If format-string is not specified, datetime-expression is truncated to the nearest day, as if 'DD' is specified for format-string.

datetime-expression
An expression that must return a value that is a DATE, a TIME, or a TIMESTAMP. String representations of these data types are not supported and must be explicitly cast to a DATE, TIME, or TIMESTAMP for use with this function; alternatively, you can use the TRUNC_TIMESTAMP function for a string representation of a date or timestamp.

format-string
An expression that returns a built-in character string data type with an actual length that is not greater than 254 bytes. The format element in format-string specifies how datetime-expression should be truncated. For example, if format-string is 'DD', a timestamp that is represented by datetime-expression is truncated to the nearest day. Leading and trailing blanks are removed from the string, and the resulting substring must be a valid format element for the type of datetime-expression (SQLSTATE 22007). The default is 'DD', which cannot be used if the data type of datetime-expression is TIME.

Allowable values for format-string are listed in the table of format elements found in the description of the ROUND function.

locale-name
A character constant that specifies the locale used to determine the first day of the week when using format element values DAY, DY, or D. The value of locale-name is not case sensitive and must be a valid locale (SQLSTATE 42815). For information on valid locales and their naming, see “Locale names for SQL and XQuery”. If locale-name is not specified, the value of the special register CURRENT LOCALE LC_TIME is used.

The result of the function has the same date type as datetime-expression. The result can be null; if any argument is null, the result is the null value.

The data type and length attribute of the result are the same as the data type and length attribute of the first argument.

The result can be null if the argument can be null or if the argument is not a decimal floating-point number and the database is configured with dft_sqlmathwarn set to YES; the result is the null value if the argument is null.

Notes

- Determinism: TRUNCATE is a deterministic function. However, the following invocations of the function depend on the value of the special register CURRENT LOCALE LC_TIME.
  - Truncate of a datetime value when locale-name is not explicitly specified and one of the following is true:
TRUNCATE or TRUNC

- `format-string` is not a constant
- `format-string` is a constant and includes format elements that are locale sensitive

Invocations of the function that depend on the value of a special register cannot be used wherever special registers cannot be used.

Examples

- Using the EMPLOYEE table, calculate the average monthly salary for the highest paid employee. Truncate the result two places to the right of the decimal point.
  
  ```sql
  SELECT TRUNCATE(MAX(SALARY)/12,2)
  FROM EMPLOYEE;
  ```

  Assuming the highest paid employee earns $52750.00 per year, the example returns 4395.83.

- Display the number 873.726 truncated 2, 1, 0, -1, and -2 decimal places, respectively.

  ```sql
  VALUES (TRUNCATE(873.726,2),
          TRUNCATE(873.726,1),
          TRUNCATE(873.726,0),
          TRUNCATE(873.726,-1),
          TRUNCATE(873.726,-2),
          TRUNCATE(873.726,-3)) ;
  ```

  This example returns 873.720, 873.700, 873.000, 870.000, 800.000, and 0.000.

- Display the decimal-floating point number 873.726 truncated 0 decimal places.

  ```sql
  VALUES (TRUNCATE(DECFLOAT(873.726),0))
  ```

  Returns the value 873.

- Set the variable vTRNK_DT with the input date rounded to the nearest month value.
  
  ```sql
  SET vTRNK_DT = TRUNC(DATE('2000-08-16'), 'MONTH');
  ```

  The value set is 2000-08-01.

- Set the host variable TRNK_TMSTMP with the current year rounded to the nearest year value.
  
  ```sql
  SET :TRNK_TMSTMP = TRUNCATE('2000-03-14-17.30.00', 'YEAR');
  ```

  The value set is 2000-01-01-00.00.00.000000.
The schema is SYSIBM.

The TYPE_ID function returns the internal type identifier of the dynamic data type of the expression.

The argument must be a user-defined structured type. (This function cannot be used as a source function when creating a user-defined function. Because it accepts any structured data type as an argument, it is not necessary to create additional signatures to support different user-defined types.)

The data type of the result of the function is INTEGER. If expression can be null, the result can be null; if expression is null, the result is the null value.

The value returned by the TYPE_ID function is not portable across databases. The value may be different, even though the type schema and type name of the dynamic data type are the same. When coding for portability, use the TYPE_SCHEMA and TYPE_NAME functions to determine the type schema and type name.

Examples:

- A table hierarchy exists having root table EMPLOYEE of type EMP and subtable MANAGER of type MGR. Another table ACTIVITIES includes a column called WHO_RESPONSIBLE that is defined as REF(EMP) SCOPE EMPLOYEE. For each reference in ACTIVITIES, display the internal type identifier of the row that corresponds to the reference.

  ```sql
  SELECT TASK, WHO_RESPONSIBLE->NAME,
         TYPE_ID(DEREF(WHO_RESPONSIBLE))
  FROM ACTIVITIES
  ```

  The DEREF function is used to return the object corresponding to the row.
The schema is SYSIBM.

The TYPE_NAME function returns the unqualified name of the dynamic data type of the expression.

The argument must be a user-defined structured type. (This function cannot be used as a source function when creating a user-defined function. Because it accepts any structured data type as an argument, it is not necessary to create additional signatures to support different user-defined types.)

The data type of the result of the function is VARCHAR(18). If expression can be null, the result can be null; if expression is null, the result is the null value. Use the TYPE_SCHEMA function to determine the schema name of the type name returned by TYPE_NAME.

Examples:

- A table hierarchy exists having root table EMPLOYEE of type EMP and subtable MANAGER of type MGR. Another table ACTIVITIES includes a column called WHO_RESPONSIBLE that is defined as REF(EMP) SCOPE EMPLOYEE. For each reference in ACTIVITIES, display the type of the row that corresponds to the reference.

  ```sql
  SELECT TASK, WHO_RESPONSIBLE->NAME,
          TYPE_NAME(DEREF(WHO_RESPONSIBLE))
          TYPE_SCHEMA(DEREF(WHO_RESPONSIBLE))
  FROM ACTIVITIES
  ```

  The DEREF function is used to return the object corresponding to the row.
The schema is SYSIBM.

The TYPE_SCHEMA function returns the schema name of the dynamic data type of the expression.

The argument must be a user-defined structured type. This function cannot be used as a source function when creating a user-defined function. Because it accepts any structured data type as an argument, it is not necessary to create additional signatures to support different user-defined types.

The data type of the result of the function is VARCHAR(128). If expression can be null, the result can be null; if expression is null, the result is the null value. Use the TYPE_NAME function to determine the type name associated with the schema name returned by TYPE_SCHEMA.
The schema is SYSIBM.

The UCASE function is identical to the TRANSLATE function except that only the first argument (char-string-exp) is specified.

UCASE is a synonym for UPPER.
UCASE (locale sensitive)

The schema is SYSIBM.

The UCASE function returns a string in which all characters have been converted to uppercase characters using the rules associated with the specified locale.

UCASE is a synonym for UPPER.
The schema is SYSIBM. (The SYSFUN version of this function continues to be available for upward compatibility.)

The UPPER function is identical to the TRANSLATE function except that only the first argument (char-string-exp) is specified.

In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.
The schema is SYSIBM.

The UPPER function returns a string in which all characters have been converted to uppercase characters using the rules associated with the specified locale.

string-expression
An expression that returns a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC string. If string-expression is CHAR or VARCHAR, the expression must not be FOR BIT DATA (SQLSTATE 42815).

locale-name
A character constant that specifies the locale that defines the rules for conversion to uppercase characters. The value of locale-name is not case sensitive and must be a valid locale (SQLSTATE 42815). For information on valid locales and their naming, see “Locale names for SQL and XQuery”.

code-units
An integer constant that specifies the number of code units in the result. If specified, code-units must be an integer between 1 and 32,672 if the result is character data, or between 1 and 16,336 if the result is graphic data (SQLSTATE 42815). If code-units is not explicitly specified, it is implicitly the length attribute of string-expression. If OCTETS is specified and the result is graphic data, the value of code-units must be even (SQLSTATE 428GC).

CODEUNITS16, CODEUNITS32, or OCTETS
Specifies the string unit of code-units.

CODEUNITS16 specifies that code-units is expressed in 16-bit UTF-16 code units. CODEUNITS32 specifies that code-units is expressed in 32-bit UTF-32 code units. OCTETS specifies that code-units is expressed in bytes.

If a string unit is not explicitly specified, the data type of the result determines the unit that is used. If the result is graphic data, code-units is expressed in two-byte units; otherwise, it is expressed in bytes. For more information about CODEUNITS16, CODEUNITS32, and OCTETS, see “String units in built-in functions” in “Character strings”.

The result of the function is VARCHAR if string-expression is CHAR or VARCHAR, and VARGRAPHIC if string-expression is GRAPHIC or VARGRAPHIC.

The length attribute of the result is determined by the implicit or explicit value of code-units, the implicit or explicit string unit, and the result data type, as shown in the following table:

<table>
<thead>
<tr>
<th>String unit</th>
<th>Character result type</th>
<th>Graphic result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CODEUNITS16</td>
<td>MIN(code-units * 3, 32768)</td>
<td>code-units</td>
</tr>
<tr>
<td>CODEUNITS32</td>
<td>MIN(code-units * 4, 32768)</td>
<td>MIN(code-units * 2, 16336)</td>
</tr>
<tr>
<td>OCTETS</td>
<td>code-units</td>
<td>MIN(code-units / 2, 16336)</td>
</tr>
</tbody>
</table>
UPPER (locale sensitive)

The actual length of the result might be greater than the length of string-expression. If the actual length of the result is greater than the length attribute of the result, an error is returned (SQLSTATE 42815). If the number of code units in the result exceeds the value of code-units, an error is returned (SQLSTATE 42815).

If string-expression is not in UTF-16, this function performs code page conversion of string-expression to UTF-16, and of the result from UTF-16 to the code page of string-expression. If either code page conversion results in at least one substitution character, the result includes the substitution character, a warning is returned (SQLSTATE 01517), and the warning flag SQLWARN8 in the SQLCA is set to ‘W’.

If the first argument can be null, the result can be null; if the first argument is null, the result is the null value.

Examples:
- Ensure that the characters in the value of column JOB in the EMPLOYEE table are returned in uppercase characters.
  ```sql
  SELECT UPPER(JOB, 'en_US')
  FROM EMPLOYEE
  WHERE EMPNO = '000020'
  
  The result is the value ‘MANAGER’.
  
  - Find the uppercase characters for all the ‘ı’ characters in a Turkish string.
    ```sql
    VALUES UPPER('İ/SM790000ıı', 'tr_TR', 0, CODEUNITS16)
    ```
    The result is the string ‘İ/SM790000ıı’.
    
  - Find the uppercase form of the German ‘ß’ (sharp S).
    ```sql
    VALUES UPPER('ß', 'de', 2, CODEUNITS16)
    ```
    The result is the string ‘SS’. Note that code-units must be specified in this example, because there are more code units in the result than in string-expression.
The schema is SYSIBM.

The VALUE function returns the first argument that is not null.

VALUE is a synonym for COALESCE.
The schema is SYSIBM. The function name cannot be specified as a qualified name when keywords are used in the function signature.

The VARCHAR function returns a varying-length character string representation of:

- An integer number, if the first argument is a SMALLINT, INTEGER, or BIGINT
- A decimal number, if the first argument is a decimal number
- A double-precision floating-point number, if the first argument is a DOUBLE or REAL
- A decimal floating-point number, if the first argument is a (DECFLOAT)
- A character string, if the first argument is any type of character string
- A graphic string (Unicode databases only), if the first argument is any type of graphic string
- A datetime value, if the first argument is a DATE, TIME, or TIMESTAMP

In a Unicode database, when the output string is truncated part-way through a multiple-byte character:
- If the input was a character string, the partial character is replaced with one or more blanks
- If the input was a graphic string, the partial character is replaced by the empty string

Do not rely on either of these behaviors, because they might change in a future release.

The result of the function is a varying-length character string. If the first argument can be null, the result can be null. If the first argument is null, the result is the null value.

**Integer to varchar**

- \textit{integer-expression}
  - An expression that returns a value that is of an integer data type (SMALLINT, INTEGER, or BIGINT).

  The result is the varying-length string representation of \textit{integer-expression} in the form of an SQL integer constant. The length attribute of the result depends on whether \textit{integer-expression} is a small, large or big integer as follows:
  - If the first argument is a small integer, the maximum length of the result is 6.
  - If the first argument is a large integer, the maximum length of the result is 11.
  - If the first argument is a big integer, the maximum length of the result is 20.

  The actual length of the result is the smallest number of characters that can be used to represent the value of the argument. Leading zeroes are not included. If the argument is negative, the first character of the result is a minus sign. Otherwise, the first character is a digit.

  The code page of the result is the code page of the section.

**Decimal to varchar**

- \textit{decimal-expression}
  - An expression that returns a value that is a decimal data type. The DECIMAL scalar function can be used to change the precision and scale.

  \textit{decimal-character}
  - Specifies the single-byte character constant that is used to delimit the decimal digits in the result character string. The character constant cannot be a digit, the plus sign (+), the minus sign (−), or a blank (SQLSTATE 42815). The default is the period (.) character.
The result is a varying-length character string representation of 
decimal-expression in the form of an SQL decimal constant. The length
attribute of the result is 2+p, where p is the precision of decimal-expression.
The actual length of the result is the smallest number of characters that can
be used to represent the result, except that trailing zeros are included.
Leading zeros are not included. If decimal-expression is negative, the first
character of the result is a minus sign; otherwise, the first character is a
digit or the decimal character. If the scale of decimal-expression is zero, the
decimal character is not returned.

The code page of the result is the code page of the section.

Floating-point to varchar

floating-point-expression
An expression that returns a value that is a floating-point data type
(DOUBLE or REAL).

decimal-character
Specifies the single-byte character constant that is used to delimit the
decimal digits in the result character string. The character constant
cannot be a digit, the plus sign (+), the minus sign (−), or a blank
(SQLSTATE 42815). The default is the period (.) character.

The result is a varying-length character string representation of
floating-point-expression in the form of an SQL floating-point constant.
The maximum length of the result is 24. The actual length of the result is
the smallest number of characters that can represent the value of
floating-point-expression such that the mantissa consists of a single digit
other than zero followed by the decimal-character and a sequence of digits.
If floating-point-expression is negative, the first character of the result is a
minus sign; otherwise, the first character is a digit. If floating-point-
expression is zero, the result is 0E0.

The code page of the result is the code page of the section.

Decimal floating-point to varchar

decimal-floating-point-expression
An expression that returns a value that is a decimal floating-point data
type (DECFLOAT).

decimal-character
Specifies the single-byte character constant that is used to delimit the
decimal digits in the result character string. The character constant
cannot be a digit, the plus sign (+), the minus sign (−), or a blank
(SQLSTATE 42815). The default is the period (.) character.

The result is a varying-length character string representation of
decimal-floating-point-expression in the form of an SQL decimal floating-point
constant. The maximum length of the result is 42. The actual length of the
result is the smallest number of characters that can represent the value of
decimal-floating-point-expression. If decimal-floating-point-expression is negative,
the first character of the result is a minus sign; otherwise, the first
character is a digit. If decimal-floating-point-expression is zero, the result is 0.

If the value of decimal-floating-point-expression is the special value Infinity,
sNaN, or NaN, the strings “INFINITY”, “SNAN”, and “NAN”,
respectively, are returned. If the special value is negative, the first character
of the result is a minus sign. The decimal floating-point special value NaN does not result in a warning when converted to a string.

The code page of the result is the code page of the section.

**Character to varchar**

character-expression

An expression that returns a value that is a built-in character string data type (CHAR, VARCHAR, or CLOB).

integer

The length attribute for the resulting varying-length character string.
The value must be between 0 and 32,762.

If the second argument is not specified:

• If the character-expression is the empty string constant, the length attribute of the result is 0.
• Otherwise, the length attribute of the result is the same as the length attribute of the first argument.

The actual length of the result is the minimum of the length attribute of the result and the actual length of character-expression. If the length of the character-expression is greater than the length attribute of the result, truncation is performed. A warning is returned (SQLSTATE 01004) unless the truncated characters were all blanks and the character-expression was not a CLOB.

**Graphic to varchar**

graphic-expression

An expression that returns a value that is a built-in graphic string data type (GRAPHIC, VARGRAPHIC, or DBCLOB).

integer

The length attribute for the resulting varying-length character string.
The value must be between 0 and 32,762.

If the second argument is not specified:

• If the graphic-expression is the empty string constant, the length attribute of the result is 0.
• Otherwise, the length attribute of the result is the same as 3 * length attribute of the first argument.

The actual length of the result is the minimum of the length attribute of the result and the actual length of graphic-expression. If the length of the graphic-expression is greater than the length attribute of the result, truncation is performed with no warning returned.

**Datetime to varchar**

datetime-expression

An expression that is of one of the following data types:

DATE  The result is the character string representation of the date in the format specified by the second argument. The length of the result is 10. An error is returned if the second argument is specified and is not a valid value (SQLSTATE 42703).

TIME  The result is the character string representation of the time in the format specified by the second argument. The length of the
result is 8. An error is returned if the second argument is specified and is not a valid value (SQLSTATE 42703).

**TIMESTAMP**

The result is the character string representation of the timestamp. If the data type of `datetime-expression` is `TIMESTAMP(0)`, the length of the result is 19. If the data type of `datetime-expression` is `TIMESTAMP(n)`, where `n` is between 1 and 12, the length of the result is `20+n`. Otherwise, the length of the result is 26. The second argument must not be specified (SQLSTATE 42815).

The code page of the result is the code page of the section.

**Notes:**
- The CAST specification should be used to increase the portability of applications when the first argument is numeric, or the first argument is a string and the length argument is specified. For more information, see “CAST specification”.
- A binary string is allowed as the first argument to the function, and the resulting varying-length string is a FOR BIT DATA character string.

**Examples**
- **Example 1:** Make EMPNO varying-length with a length of 10.
  ```sql
  SELECT VARCHAR(EMPNO, 10)
  INTO :VARHV
  FROM EMPLOYEE
  ```

- **Example 2:** Set the host variable JOB_DESC, defined as VARCHAR(8), to the VARCHAR equivalent of the job description (which is the value of the JOB column), defined as CHAR(8), for employee Dolores Quintana.
  ```sql
  SELECT VARCHAR(JOB)
  INTO :JOB_DESC
  FROM EMPLOYEE
  WHERE LASTNAME = 'QUINTANA'
  ```

- **Example 3:** The EDLEVEL column is defined as SMALLINT. The following returns the value as a varying-length character string.
  ```sql
  SELECT VARCHAR(EDLEVEL)
  FROM EMPLOYEE
  WHERE LASTNAME = 'HAAS'
  ```

  Results in the value ‘18’.

- **Example 4:** The SALARY and COMM columns are defined as DECIMAL with a precision of 9 and a scale of 2. Return the total income for employee Haas using the comma decimal character.
  ```sql
  SELECT VARCHAR(SALARY + COMM, ',')
  FROM EMPLOYEE
  WHERE LASTNAME = 'HAAS'
  ```

  Results in the value ‘56970,00’.
The schema is SYSIBM.

The VARCHAR_BIT_FORMAT function returns a bit string representation of a character string that has been formatted using a character template. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

**character-expression**

An expression that returns a value that is a built-in character string that is not a CLOB (SQLSTATE 42815). The required length is determined by the specified format string and how the value is interpreted.

**format-string**

A character constant that contains a template for how the bytes of character-expression are to be interpreted.

Valid format strings include: 'xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx' and 'XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX' (SQLSTATE 42815) where each 'x' or 'X' corresponds to one hexadecimal digit in the result.

The result of the function is a varying-length character string FOR BIT DATA with the length attribute and actual length based on the format string. For the two valid format strings listed above, the length attribute of the result is 36 and the actual length is 16. If the first argument can be null, the result can be null; if the first argument is null, the result is the null value.

**Examples**

- Represent a Universal Unique Identifier in its binary form:
  ```sql
  VARCHAR_BIT_FORMAT('d83d6360-1818-11db-9804-b622a1ef5492',
  'xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx')
  ```
  Result returned:
  x'D83D6360181811DB9804B622A1EF5492'

- Represent a Universal Unique Identifier in its binary form:
  ```sql
  VARCHAR_BIT_FORMAT('08306360-1818-11db-9804-b622a1ef5492',
  'XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX')
  ```
  Result returned:
  x'D83D6360181811DB9804B622A1EF5492'
The VARCHAR_FORMAT function returns a character string based on applying the specified format string argument, if provided, to the value of the first argument. If any argument of the VARCHAR_FORMAT function can be null, the result can be null; if any argument is null, the result is the null value.

The expression must be formatted according to a specified character template.

**Character to varchar**

\[\text{VARCHAR\_FORMAT}\left(\text{character-expression}\right)\]

**Timestamp to varchar**

\[\text{VARCHAR\_FORMAT}\left(\text{timestamp-expression}, \text{format-string}, \text{locale-name}\right)\]

**Decimal floating-point to varchar**

\[\text{VARCHAR\_FORMAT}\left(\text{decimal-floating-point-expression}, \text{format-string}\right)\]

The schema is SYSIBM.

The VARCHAR_FORMAT function returns a character string based on applying the specified format string argument, if provided, to the value of the first argument. If any argument of the VARCHAR_FORMAT function can be null, the result can be null; if any argument is null, the result is the null value.

The expression must be formatted according to a specified character template.

**Character to varchar**

*character-expression*

An expression that returns a value that must be a built-in CHAR or VARCHAR data type. In a Unicode database, if a supplied argument is a GRAPHIC or VARGRAPHIC data type, it is first converted to VARCHAR before evaluating the function.

The result is a VARCHAR with a length attribute that matches the length attribute of the argument. The value of the result is the same as the value of *character-expression*.

**Timestamp to varchar**

*timestamp-expression*

An expression that returns a value that must be a DATE or TIMESTAMP, or a valid string representation of a date or timestamp that is not a CLOB or DBCLOB. If the argument is a string, the format-string argument must also be specified. In a Unicode database, if a supplied argument is a graphic string representation of a data, time, or timestamp, it is first converted to a character string before evaluating the function.

If *timestamp-expression* is a DATE or a valid string representation of a date, it is first converted to a TIMESTAMP(0) value, assuming a time of exactly midnight (00.00.00).

For the valid formats of string representations of datetime values, see “String representations of datetime values” in “Datetime values”.

Chapter 3. Functions  611
format-string

The expression must return a value that is a built-in CHAR, VARCHAR, numeric, or datetime data type. If the value is not a CHAR or VARCHAR data type, it is implicitly cast to VARCHAR before evaluating the function. In a Unicode database, if the supplied argument is a GRAPHIC or VARGRAPHIC data type, it is first converted to VARCHAR before evaluating the function. The actual length must not be greater than 254 bytes (SQLSTATE 22007). The value is a template for how timestamp-expression is to be formatted.

A valid format-string must contain a combination of the format elements listed below (SQLSTATE 22007). Two format elements can optionally be separated by one or more of the following separator characters:

- minus sign (-)
- period (.)
- slash (/)
- comma (,)
- apostrophe (')
- semi-colon (;)
- colon (:)
- blank ( )

Separator characters can also be specified at the start or end of format-string.

Table 61. Format elements for the VARCHAR_FORMAT function

<table>
<thead>
<tr>
<th>Format element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AM or PM</td>
<td>Meridian indicator (morning or evening) without periods. This format element is dependent on locale-name, if specified; otherwise, it is dependent on the value of the special register CURRENT LOCALE LC_TIME.</td>
</tr>
<tr>
<td>A.M. or P.M.</td>
<td>Meridian indicator (morning or evening) with periods. This format element uses the exact strings ‘A.M.’ or ‘P.M.’ and is independent of the locale name in effect.</td>
</tr>
<tr>
<td>CC</td>
<td>Century (01-99). If the last two digits of the four-digit year are zero, the result is the first two digits of the year; otherwise, the result is the first two digits of the year plus one.</td>
</tr>
<tr>
<td>DAY, Day, or day</td>
<td>Name of the day in uppercase, titlecase, or lowercase format. The language used is dependent on locale-name, if specified; otherwise, it is dependent on the value of the special register CURRENT LOCALE LC_TIME.</td>
</tr>
<tr>
<td>DY, Dy, or dy</td>
<td>Abbreviated name of the day in uppercase, titlecase, or lowercase format. The language used is dependent on locale-name, if specified; otherwise, it is dependent on the value of the special register CURRENT LOCALE LC_TIME.</td>
</tr>
</tbody>
</table>
Table 61. Format elements for the VARCHAR_FORMAT function (continued)

<table>
<thead>
<tr>
<th>Format element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>Day of the week (1-7). The first day of the week is dependent on locale-name, if specified; otherwise, it is dependent on the value of the special register CURRENT LOCALE LC_TIME.</td>
</tr>
<tr>
<td>DD</td>
<td>Day of month (01-31).</td>
</tr>
<tr>
<td>DDD</td>
<td>Day of year (001-366).</td>
</tr>
<tr>
<td>FF or FF\text{n}</td>
<td>Fractional seconds (0-999999999999). The number ( n ) is used to specify the number of digits expected in the string-expression. Valid values for ( n ) are 1-12 with no leading zeros. Specifying FF is equivalent to specifying FF6. If the timestamp precision of timestamp-expression is less than what is specified by the format, zero digits are padded onto the right of the specified digits.</td>
</tr>
<tr>
<td>HH</td>
<td>HH behaves the same as HH12.</td>
</tr>
<tr>
<td>HH12</td>
<td>Hour of the day (01-12) in 12-hour format.</td>
</tr>
<tr>
<td>HH24</td>
<td>Hour of the day (00-24) in 24-hour format.</td>
</tr>
<tr>
<td>IW</td>
<td>ISO week of the year (01-53). The week starts on Monday and includes seven days. Week 1 is the first week of the year to contain a Thursday, which is equivalent to the first week of the year to contain January 4.</td>
</tr>
<tr>
<td>I</td>
<td>ISO year (0-9). The last digit of the year based on the ISO week that is returned.</td>
</tr>
<tr>
<td>IY</td>
<td>ISO year (00-99). The last two digits of the year based on the ISO week that is returned.</td>
</tr>
<tr>
<td>IYY</td>
<td>ISO year (000-999). The last three digits of the year based on the ISO week that is returned.</td>
</tr>
<tr>
<td>IYYYY</td>
<td>ISO year (0000-9999). The 4-digit year based on the ISO week that is returned.</td>
</tr>
<tr>
<td>J</td>
<td>Julian day (number of days since January 1, 4713 BC).</td>
</tr>
<tr>
<td>MI</td>
<td>Minute (00-59).</td>
</tr>
<tr>
<td>MM</td>
<td>Month (01-12).</td>
</tr>
<tr>
<td>NNNNNNN</td>
<td>Microseconds (000000-999999). Same as FF6.</td>
</tr>
<tr>
<td>MONTH, Month, or month</td>
<td>Name of the month in uppercase, titlecase, or lowercase format. The language used is dependent on locale-name, if specified; otherwise, it is dependent on the value of the special register CURRENT LOCALE LC_TIME.</td>
</tr>
<tr>
<td>Format element</td>
<td>Description</td>
</tr>
<tr>
<td>----------------</td>
<td>-------------</td>
</tr>
<tr>
<td>MON, Mon, or mon</td>
<td>Abbreviated name of the month in uppercase, titlecase, or lowercase format. The language used is dependent on <code>locale-name</code>, if specified; otherwise, it is dependent on the value of the special register CURRENT LOCALE LC_TIME.</td>
</tr>
<tr>
<td>Q</td>
<td>Quarter (1-4), where the months January through March return 1.</td>
</tr>
<tr>
<td>RR</td>
<td>RR behaves the same as YY.</td>
</tr>
<tr>
<td>RRRR</td>
<td>RRRR behaves the same as YYYY.</td>
</tr>
<tr>
<td>SS</td>
<td>Seconds (00-59).</td>
</tr>
<tr>
<td>SSSSS</td>
<td>Seconds since previous midnight (00000-86400).</td>
</tr>
<tr>
<td>W</td>
<td>Week of the month (1-5), where week 1 starts on the first day of the month and ends on the seventh day.</td>
</tr>
<tr>
<td>WW</td>
<td>Week of the year (01-53), where week 1 starts on January 1 and ends on January 7.</td>
</tr>
<tr>
<td>Y</td>
<td>Last digit of the year (0-9).</td>
</tr>
<tr>
<td>YY</td>
<td>Last two digits of the year (00-99).</td>
</tr>
<tr>
<td>YYY</td>
<td>Last three digits of the year (000-999).</td>
</tr>
<tr>
<td>YYYY</td>
<td>4-digit year (0000-9999).</td>
</tr>
</tbody>
</table>

**Note:** The format elements in Table 61 on page 612 are not case sensitive, except for the following:

- AM, PM
- A.M., P.M.
- DAY, Day, day
- DY, Dy, dy
- D
- MONTH, Month, month
- MON, Mon, mon

In cases where format elements are ambiguous, the case insensitive format elements will be considered first. For example, DDYYYY would be interpreted as DD followed by YYYY, rather than D followed by DY, followed by YY.

If `format-string` is not specified, the format used is based on the value of the special register CURRENT LOCALE LC_TIME.

**locale-name**

A character constant that specifies the locale used for the following format elements:

- AM, PM
- DAY, Day, day
- DY, Dy, dy
- D
The value of `locale-name` is not case sensitive and must be a valid locale (SQLSTATE 42815). For information on valid locales and their naming, see “Locale names for SQL and XQuery” in the Globalization Guide. If `locale-name` is not specified, the value of the special register CURRENT LOCALE LC_TIME is used.

The result is a representation of `timestamp-expression` in the format specified by `format-string`. The `format-string` is interpreted as a series of format elements that can optionally be separated by one or more separator characters. A string of characters in `format-string` is interpreted as the longest matching format element in Table 61 on page 612. If two format elements containing the same characters are not delimited by a separator character, the specification is interpreted, starting from the left, as the longest matching format element in the table, and continues until matches are found for the remainder of the format string. For example, ‘YYYYYYDD’ is interpreted as the format elements ‘YYYY’, ‘YY’, and ‘DD’.

The result is a varying-length character string. The length attribute is 254. The `format-string` determines the actual length of the result. If the resulting string exceeds the length attribute of the result, the result is truncated.

**Decimal floating-point to varchar**

`decimal-floating-point-expression`

An expression that returns a value of any built-in numeric data type. If the argument is not a decimal floating-point value, it is converted to DECFLOAT(34) for processing.

`format-string`

The expression must return a value that is a built-in CHAR, VARCHAR, numeric, or datetime data type. If the value is not a CHAR or VARCHAR data type, it is implicitly cast to VARCHAR before evaluating the function. In a Unicode database, if the supplied argument is a GRAPHIC or VARGRAPHIC data type, it is first converted to VARCHAR before evaluating the function. The actual length must not be greater than 254 bytes (SQLSTATE 22018). The value is a template for how `decimal-floating-point-expression` is to be formatted. Format elements specified as a prefix can be used only at the beginning of the template. Format elements specified as a suffix can be used only at the end of the template. The format elements are case sensitive. The template must not contain more than one of the MI, S, or PR format elements (SQLSTATE 22018).

<table>
<thead>
<tr>
<th>Format element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Each 0 represents a significant digit. Leading zeros in a number are displayed as zeros.</td>
</tr>
<tr>
<td>9</td>
<td>Each 9 represents a significant digit. Leading zeros in a number are displayed as blanks.</td>
</tr>
<tr>
<td>MI</td>
<td>Suffix: If <code>decimal-floating-point-expression</code> is a negative number, a trailing minus sign (−) is included in the result. If <code>decimal-floating-point-expression</code> is a positive number, a trailing blank is included in the result.</td>
</tr>
</tbody>
</table>
### VARCHAR_FORMAT

**Table 62. Format elements for the VARCHAR_FORMAT function (continued)**

<table>
<thead>
<tr>
<th>Format element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>Prefix: If <code>decimal-floating-point-expression</code> is a negative number, a leading minus sign (−) is included in the result. If <code>decimal-floating-point-expression</code> is a positive number, a leading plus sign (+) is included in the result.</td>
</tr>
<tr>
<td>PR</td>
<td>Suffix: If <code>decimal-floating-point-expression</code> is a negative number, a leading less than character (&lt;) and a trailing greater than character (&gt;) are included in the result. If <code>decimal-floating-point-expression</code> is a positive number, a leading space and a trailing space are included in the result.</td>
</tr>
<tr>
<td>$</td>
<td>Prefix: A leading dollar sign ($) is included in the result.</td>
</tr>
<tr>
<td>,</td>
<td>Specifies that a comma be included in that location in the result. This comma is used as a group separator.</td>
</tr>
<tr>
<td>.</td>
<td>Specifies that a period be included in that location in the result. This period is used as a decimal point.</td>
</tr>
</tbody>
</table>

If `format-string` is not specified, `decimal-floating-point-expression` is formatted in the form of an SQL decimal floating-point constant. If `decimal-floating-point-expression` is negative, the first character of the result is a minus sign; otherwise, the first character is a digit. If `decimal-floating-point-expression` is zero, the result is 0.

The result is a varying-length character string representation of `decimal-floating-point-expression`. The length attribute is 254. The actual length of the result is determined by the `format-string`, if specified; otherwise, the actual length of the result is the smallest number of characters that can represent the value of `decimal-floating-point-expression`. If the resulting string exceeds the length attribute of the result, the result will be truncated.

If the value of `decimal-floating-point-expression` is the special value Infinity, sNaN, or NaN, the strings "INFINITY", "SNAN", and "NAN", respectively, are returned. If the special value is negative, the first character of the result is a minus sign. The decimal floating-point special value sNaN does not result in an exception when converted to a string.

If `format-string` does not include any of the format elements MI, S, or PR, and the value of `decimal-floating-point-expression` is negative, then a minus sign (−) will be included in the result; otherwise, a blank will be included in the result.

If the number of digits to the left of the decimal point in `decimal-floating-point-expression` is greater than the number of digits to the left of the decimal point in `format-string`, the result is a string of number sign (#) characters. If the number of digits to the right of the decimal point in `decimal-floating-point-expression` is greater than the number of digits to the right of the decimal point in `format-string`, the result is `decimal-floating-point-expression` rounded to the number of digits to the right of the decimal point in `format-string`. The DECFLOAT rounding mode will not be used. The rounding behavior of VARCHAR_FORMAT corresponds to a value of ROUND_HALF_UP.

The code page of the result is the code page of the section.
Notes:

- **Julian and Gregorian calendar**: For Timestamp to varchar, the transition from the Julian calendar to the Gregorian calendar on 15 October 1582 is taken into account by this function.

- **Determinism**: VARCHAR_FORMAT is a deterministic function. However, the following invocations of the function depend on the value of the special register CURRENT LOCALE LC_TIME:
  - Timestamp to varchar, when format-string is not explicitly specified, or when locale-name is not explicitly specified and one of the following is true:
    - format-string is not a constant
    - format-string is a constant and includes format elements that are locale sensitive

  These invocations that depend on the value of a special register cannot be used wherever special registers cannot be used (SQLSTATE 42621 or 428EC).

- **Syntax alternatives**: TO_CHAR is a synonym for VARCHAR_FORMAT.

Examples

- Display the table names and creation timestamps for all of the system tables whose name starts with 'SYSU'.

  ```sql
  SELECT VARCHAR(TABNAME, 20) AS TABLE_NAME,
  VARCHAR_FORMAT(CREATE_TIME, 'YYYY-MM-DD HH24:MI:SS') AS CREATION_TIME
  FROM SYSUTABLES
  WHERE TABNAME LIKE 'SYSU'
  ```

  This example returns the following:

<table>
<thead>
<tr>
<th>TABLE_NAME</th>
<th>CREATION_TIME</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYSUSERAUTH</td>
<td>2000-05-19 08:18:56</td>
</tr>
<tr>
<td>SYSUSEROPTIONS</td>
<td>2000-05-19 08:18:56</td>
</tr>
</tbody>
</table>

- Assume that the variable TMSTAMP is defined as a TIMESTAMP and has the following value: 2007-03-09-14.07.38.123456. The following examples show several invocations of the function and the resulting string values. The result data type in each case is VARCHAR(254).

<table>
<thead>
<tr>
<th>Function invocation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>VARCHAR_FORMAT(TMSTAMP, 'YYYYMMDDHHMISSFF3')</td>
<td>20070309020738123</td>
</tr>
<tr>
<td>VARCHAR_FORMAT(TMSTAMP, 'YYYYMMDDHH24MISS')</td>
<td>20070309140738</td>
</tr>
<tr>
<td>VARCHAR_FORMAT(TMSTAMP, 'YYYYMMDDHHMMI')</td>
<td>200703090207</td>
</tr>
<tr>
<td>VARCHAR_FORMAT(TMSTAMP, 'DD/MM/YY')</td>
<td>09/03/07</td>
</tr>
<tr>
<td>VARCHAR_FORMAT(TMSTAMP, 'MM-DD-YYYY')</td>
<td>03-09-2007</td>
</tr>
<tr>
<td>VARCHAR_FORMAT(TMSTAMP, 'J')</td>
<td>2454169</td>
</tr>
<tr>
<td>VARCHAR_FORMAT(TMSTAMP, 'Q')</td>
<td>1</td>
</tr>
<tr>
<td>VARCHAR_FORMAT(TMSTAMP, 'W')</td>
<td>2</td>
</tr>
<tr>
<td>VARCHAR_FORMAT(TMSTAMP, 'IW')</td>
<td>10</td>
</tr>
<tr>
<td>VARCHAR_FORMAT(TMSTAMP, 'Ww')</td>
<td>10</td>
</tr>
<tr>
<td>VARCHAR_FORMAT(TMSTAMP, 'Month', 'en_US')</td>
<td>March</td>
</tr>
<tr>
<td>VARCHAR_FORMAT(TMSTAMP, 'MONTH', 'en_US')</td>
<td>MARCH</td>
</tr>
<tr>
<td>VARCHAR_FORMAT(TMSTAMP, 'MON', 'en_US')</td>
<td>MAR</td>
</tr>
<tr>
<td>VARCHAR_FORMAT(TMSTAMP, 'Day', 'en_US')</td>
<td>Friday</td>
</tr>
<tr>
<td>VARCHAR_FORMAT(TMSTAMP, 'DAY', 'en_US')</td>
<td>FRIDAY</td>
</tr>
<tr>
<td>VARCHAR_FORMAT(TMSTAMP, 'Dy', 'en_US')</td>
<td>Fri</td>
</tr>
<tr>
<td>VARCHAR_FORMAT(TMSTAMP, 'Month', 'de.DE')</td>
<td>März</td>
</tr>
<tr>
<td>VARCHAR_FORMAT(TMSTAMP, 'MONTH', 'de.DE')</td>
<td>MÄRZ</td>
</tr>
<tr>
<td>VARCHAR_FORMAT(TMSTAMP, 'MON', 'de.DE')</td>
<td>MRZ</td>
</tr>
<tr>
<td>VARCHAR_FORMAT(TMSTAMP, 'Day', 'de.DE')</td>
<td>Freitag</td>
</tr>
<tr>
<td>VARCHAR_FORMAT(TMSTAMP, 'DAY', 'de.DE')</td>
<td>FREITAG</td>
</tr>
<tr>
<td>VARCHAR_FORMAT(TMSTAMP, 'Dy', 'de.DE')</td>
<td>Fr</td>
</tr>
</tbody>
</table>
**VARCHAR_FORMAT**

- Assume that the variable DTE is defined as a DATE and has the following value: 2007-03-09. The following examples show several invocations of the function and the resulting string values. The result data type in each case is VARCHAR(254).

<table>
<thead>
<tr>
<th>Function invocation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>VARCHAR_FORMAT(DTE,'YYYYMMDD')</td>
<td>20070309</td>
</tr>
<tr>
<td>VARCHAR_FORMAT(DTE,'YYYYMMDDHH24MISS')</td>
<td>20070309000000</td>
</tr>
</tbody>
</table>

- Assume that the variables POSNUM and NEGNUM are defined as DECFLOAT(34) and have the following values: 1234.56 and -1234.56, respectively. The following examples show several invocations of the function and the resulting string values. The result data type in each case is VARCHAR(254).

<table>
<thead>
<tr>
<th>Function invocation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>VARCHAR_FORMAT(POSNUM)</td>
<td>'1234.56'</td>
</tr>
<tr>
<td>VARCHAR_FORMAT(NEGNUM)</td>
<td>'-1234.56'</td>
</tr>
<tr>
<td>VARCHAR_FORMAT(POSNUM, '9999.99')</td>
<td>'1234.56'</td>
</tr>
<tr>
<td>VARCHAR_FORMAT(NEGNUM, '9999.99')</td>
<td>'-1234.56'</td>
</tr>
<tr>
<td>VARCHAR_FORMAT(POSNUM, '99999.99')</td>
<td>' 1234.56'</td>
</tr>
<tr>
<td>VARCHAR_FORMAT(NEGNUM, '99999.99')</td>
<td>' 1234.56'</td>
</tr>
<tr>
<td>VARCHAR_FORMAT(POSNUM, '00000.00')</td>
<td>'01234.56'</td>
</tr>
<tr>
<td>VARCHAR_FORMAT(NEGNUM, '00000.00')</td>
<td>'01234.56'</td>
</tr>
<tr>
<td>VARCHAR_FORMAT(POSNUM, '9999.99MI')</td>
<td>'1234.56-'</td>
</tr>
<tr>
<td>VARCHAR_FORMAT(NEGNUM, '9999.99MI')</td>
<td>'&lt;1234.56&gt;'</td>
</tr>
<tr>
<td>VARCHAR_FORMAT(POSNUM, 'S9999.99')</td>
<td>'+1234.56'</td>
</tr>
<tr>
<td>VARCHAR_FORMAT(NEGNUM, 'S9999.99')</td>
<td>'-1234.56'</td>
</tr>
<tr>
<td>VARCHAR_FORMAT(POSNUM, '9999.99PR')</td>
<td>'+1234.56'</td>
</tr>
<tr>
<td>VARCHAR_FORMAT(NEGNUM, '9999.99PR')</td>
<td>'&lt;1234.56&gt;'</td>
</tr>
<tr>
<td>VARCHAR_FORMAT(POSNUM, 'S$9,999.99')</td>
<td>'+$1,234.56'</td>
</tr>
<tr>
<td>VARCHAR_FORMAT(NEGNUM, 'S$9,999.99')</td>
<td>'-$1,234.56'</td>
</tr>
</tbody>
</table>
The schema is SYSIBM.

The VARCHAR_FORMAT_BIT function returns a character representation of a bit string that has been formatted using a character template.

**bit-data-expression**

An expression that returns a value that is a built-in character-string FOR BIT DATA data type (SQLSTATE 42815). The required length is determined by the specified format string and how the value is interpreted.

**format-string**

A character constant that contains a template for how the result is to be formatted.

Valid format strings include: 'xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx' and 'XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX' (SQLSTATE 42815) where each 'x' or 'X' corresponds to one hexadecimal digit from **bit-data-expression**.

The result of the function is a varying-length character string with the length attribute and actual length based on the format string. For the two valid format strings listed above, the length attribute is 36 and the actual length is 36 bytes. If the first argument can be null, the result can be null; if the first argument is null, the result is the null value.

**Examples**

- Represent a Universal Unique Identifier in its formatted form:
  
  ```sql
  VARCHAR_FORMAT_BIT(CAST(x'd83d6360181811db9804b622a1ef5492' AS VARCHAR(16) FOR BIT DATA), 'xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx')
  ```

  Result returned:
  'd83d6360-1818-11db-9804-b622a1ef5492'

- Represent a Universal Unique Identifier in its formatted form:
  
  ```sql
  VARCHAR_FORMAT_BIT(CAST(x'd83d6360181811db9804b622a1ef5492' AS CHAR(16) FOR BIT DATA), 'XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX')
  ```

  Result returned:
  'D83D6360-1818-11DB-9804-B622A1EF5492'
The schema is SYSIBM. The function name cannot be specified as a qualified name when keywords are used in the function signature.

The VARGRAPHIC function returns a varying-length graphic string representation of:

- An integer number (Unicode database only), if the first argument is a SMALLINT, INTEGER, or BIGINT
- A decimal number (Unicode database only), if the first argument is a decimal number
• A double-precision floating-point number (Unicode database only), if the first argument is a DOUBLE or REAL
• A decimal floating-point number (Unicode database only), if the first argument is a decimal floating-point number (DECFLOAT)
• A character string, converting single-byte characters to double-byte characters, if the first argument is any type of character string
• A graphic string, if the first argument is any type of graphic string
• A datetime value (Unicode databases only), if the first argument is a DATE, TIME, or TIMESTAMP

In a Unicode database, if a supplied argument is a character string, it is first converted to a graphic string before the function is executed. When the output string is truncated, such that the last character is a high surrogate, that surrogate is converted to the blank character (X'0020'). Do not rely on this behavior, because it might change in a future release.

The result of the function is a varying-length graphic string (VARGRAPHIC data type). If the first argument can be null, the result can be null; if the first argument is null, the result is the null value.

**Integer to vargraphic**

```
integer-expression
```

An expression that returns a value that is of an integer data type (SMALLINT, INTEGER, or BIGINT).

The result is the varying-length graphic string representation of `integer-expression` in the form of an SQL integer constant. The length attribute of the result depends on whether `integer-expression` is a small, large or big integer as follows:

- If the first argument is a small integer, the maximum length of the result is 6.
- If the first argument is a large integer, the maximum length of the result is 11.
- If the first argument is a big integer, the maximum length of the result is 20.

The actual length of the result is the smallest number of double-byte characters that can be used to represent the value of the argument. Leading zeroes are not included. If the argument is negative, the first double-byte character of the result is a minus sign; otherwise, the first double-byte character is a digit.

The code page of the result is the DBCS code page of the section.

**Decimal to vargraphic**

```
decimal-expression
decimal-character
```

An expression that returns a value that is a decimal data type. The DECIMAL scalar function can be used to change the precision and scale.

Specifies the double-byte character constant that is used to delimit the decimal digits in the result graphic string. The double-byte character constant cannot be a digit, the plus sign (+), the minus sign (−), or a blank (SQLSTATE 42815). The default is the period (.) character.
The result is a varying-length graphic string representation of
\textit{decimal-expression} in the form of an SQL decimal constant. The length
attribute of the result is $2+p$, where $p$ is the precision of \textit{decimal-expression}.
The actual length of the result is the smallest number of double-byte
characters that can be used to represent the result, except that trailing zeros
are included. Leading zeros are not included. If \textit{decimal-expression} is
negative, the first double-byte character of the result is a minus sign;
otherwise, the first double-byte character is a digit or the decimal character.
If the scale of \textit{decimal-expression} is zero, the decimal character is not
returned.

The code page of the result is the DBCS code page of the section.

\textbf{Floating-point to \texttt{vargraphic}}

\textit{floating-point-expression}
\hspace{1em}An expression that returns a value that is a floating-point data type
\hspace{1em}(DOUBLE or REAL).

\textit{decimal-character}
\hspace{1em}Specifies the double-byte character constant that is used to delimit the
decimal digits in the result graphic string. The double-byte character
constant cannot be a digit, the plus sign (+), the minus sign (−), or a
blank (SQLSTATE 42815). The default is the period (.) character.

The result is a varying-length graphic string representation of
\textit{floating-point-expression} in the form of an SQL floating-point constant.
The maximum length of the result is 24. The actual length of the result is
the smallest number of double-byte characters that can represent the value
of \textit{floating-point-expression} such that the mantissa consists of a single digit
other than zero followed by the \textit{decimal-character} and a sequence of digits.
If \textit{floating-point-expression} is negative, the first double-byte character of the
result is a minus sign; otherwise, the first double-byte character is a digit.
If \textit{floating-point-expression} is zero, the result is 0E0.

The code page of the result is the DBCS code page of the section.

\textbf{Decimal floating-point to \texttt{vargraphic}}

\textit{decimal-floating-point-expression}
\hspace{1em}An expression that returns a value that is a decimal floating-point data
\hspace{1em}type (DECFLOAT).

\textit{decimal-character}
\hspace{1em}Specifies the double-byte character constant that is used to delimit the
decimal digits in the result graphic string. The double-byte character
constant cannot be a digit, the plus sign (+), the minus sign (−), or a
blank (SQLSTATE 42815). The default is the period (.) character.

The result is a varying-length graphic string representation of
\textit{decimal-floating-point-expression} in the form of an SQL decimal floating-point
constant. The maximum length of the result is 42. The actual length of the
result is the smallest number of double-byte characters that can represent
the value of \textit{decimal-floating-point-expression}. If \textit{decimal-floating-point-
expression} is negative, the first double-byte character of the result is a
minus sign; otherwise, the first double-byte character is a digit. If
\textit{decimal-floating-point-expression} is zero, the result is 0.
If the value of *decimal-floating-point-expression* is the special value Infinity, sNaN, or NaN, the strings G'INFINITY', G'SNAN', and G'NAN', respectively, are returned. If the special value is negative, the first double-byte character of the result is a minus sign. The decimal floating-point special value sNaN does not result in a warning when converted to a string.

The code page of the result is the DBCS code page of the section.

**Character to vargraphic**

*character-expression*

An expression that returns a value that is a built-in character string data type (CHAR, VARCHAR, or CLOB).

*integer*

The length attribute for the resulting varying-length graphic string. The value must be between 0 and 16 336. If the second argument is not specified:

- If the *character-expression* is the empty string constant, the length attribute of the result is 0.
- Otherwise, the length attribute of the result is the same as the length attribute of the first argument.

The actual length of the result is the minimum of the length attribute of the result and the actual length of *character-expression*. If the length of the *character-expression* is greater than the length attribute of the result, truncation is performed with no warning returned.

Each single-byte character in *character-expression* is converted to its equivalent double-byte representation or to the double-byte substitution character in the result. Each double-byte character in *character-expression* is mapped without additional conversion. If the first byte of a double-byte character appears as the last byte of *character-expression*, it is converted to the double-byte substitution character. The sequential order of the characters in *character-expression* is preserved.

For a Unicode database, this function converts the character string from the code page of the argument to UCS-2. Every character of the argument, including double-byte characters, is converted. If a value for the second argument is provided, it specifies the required length of the resulting string (in UCS-2 characters).

The conversion to double-byte code points by the VARGRAPHIC function is based on the code page of the argument.

Double-byte characters of the argument are not converted. All other characters are converted to their corresponding double-byte equivalents. If there is no corresponding double-byte equivalent, the double-byte substitution character for the code page is used.

No warning or error code is generated if one or more double-byte substitution characters are returned in the result.

**Graphic to vargraphic**

*graphic-expression*

An expression that returns a value that is a built-in graphic string data type (GRAPHIC, VARGRAPHIC, or DBCLOB).
VARGRAPHIC

integer
The length attribute for the resulting varying-length graphic string. The value must be between 0 and 16336. If the second argument is not specified:

• If the graphic-expression is the empty string constant, the length attribute of the result is 0.
• Otherwise, the length attribute of the result is the same as the length attribute of the first argument.

The actual length of the result is the minimum of the length attribute of the result and the actual length of graphic-expression. If the length of the graphic-expression is greater than the length attribute of the result, truncation is performed. A warning is returned (SQLSTATE 01004) unless the truncated characters were all blanks and the graphic-expression was not a DBCLOB.

If the length of the graphic expression is greater than the length attribute of the result, the result is truncated. A warning is returned (SQLSTATE 01004), unless the truncated characters were all blanks, and the graphic expression was not a DBCLOB.

Datetime to vargraphic
datetime-expression
An expression that is one of the following data types:

DATE The result is the graphic string representation of the date in the format specified by the second argument. The length of the result is 10. An error is returned if the second argument is specified and is not a valid value (SQLSTATE 42703).

TIME The result is the graphic string representation of the time in the format specified by the second argument. The length of the result is 8. An error is returned if the second argument is specified and is not a valid value (SQLSTATE 42703).

TIMESTAMP
The result is the graphic string representation of the timestamp. If the data type of datetime-expression is TIMESTAMP(0), the length of the result is 19. If the data type of datetime-expression is TIMESTAMP(n), where n is between 1 and 12, the length of the result is 20+n. Otherwise, the length of the result is 26. The second argument must not be specified (SQLSTATE 42815).

The code page of the result is the DBCS code page of the section.

Note: The CAST specification should be used to increase the portability of applications when the first argument is numeric, or if the first argument is a string and the length argument is specified. For more information, see “CAST specification”.

Examples
• The EDLEVEL column is defined as SMALLINT. The following returns the value as a varying-length graphic string.

SELECT VARGRAPHIC(EDLEVEL)
FROM EMPLOYEE
WHERE LASTNAME = 'HAAS'
The SALARY and COMM columns are defined as DECIMAL with a precision of 9 and a scale of 2. Return the total income for employee Haas using the comma decimal character.

```sql
SELECT VARGRAPHIC(SALARY + COMM, ',')
FROM EMPLOYEE
WHERE LASTNAME = 'HAAS'
```

Results in the value G’56970,00’.
WEEK

The schema is SYSFUN.

The WEEK scalar function returns the week of the year of the argument as an integer value in range 1-54. The week starts with Sunday.

The argument must be a DATE, TIMESTAMP, or a valid character string representation of a date or timestamp that is not a CLOB. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

The result of the function is INTEGER. The result can be null; if the argument is null, the result is the null value.
WEEK_ISO

The schema is SYSFUN.

The WEEK_ISO function returns the week of the year of the argument as an integer value in the range 1-53. The week starts with Monday and always includes 7 days. Week 1 is the first week of the year to contain a Thursday, which is equivalent to the first week containing January 4. It is therefore possible to have up to 3 days at the beginning of a year appear in the last week of the previous year. Conversely, up to 3 days at the end of a year may appear in the first week of the next year.

The argument must be a DATE, TIMESTAMP, or a valid character string representation of a date or timestamp that is not a CLOB. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

The result of the function is INTEGER. The result can be null; if the argument is null, the result is the null value.

Example

The following list shows examples of the result of WEEK_ISO and DAYOFWEEK_ISO.

<table>
<thead>
<tr>
<th>DATE</th>
<th>WEEK_ISO</th>
<th>DAYOFWEEK_ISO</th>
</tr>
</thead>
<tbody>
<tr>
<td>1997-12-28</td>
<td>52</td>
<td>7</td>
</tr>
<tr>
<td>1997-12-31</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>1998-01-01</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>1999-01-01</td>
<td>53</td>
<td>5</td>
</tr>
<tr>
<td>1999-01-04</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1999-12-31</td>
<td>52</td>
<td>5</td>
</tr>
<tr>
<td>2000-01-01</td>
<td>52</td>
<td>6</td>
</tr>
<tr>
<td>2000-01-03</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
XMLATTRIBUTES

XMLATTRIBUTES

\[\text{XMLATTRIBUTES} = \text{attribute-value-expression} \text{ AS attribute-name}\]

The schema is SYSIBM. The function name cannot be specified as a qualified name.

The XMLATTRIBUTES function constructs XML attributes from the arguments. This function can only be used as an argument of the XMLELEMENT function. The result is an XML sequence containing an XQuery attribute node for each non-null input value.

**attribute-value-expression**

An expression whose result is the attribute value. The data type of attribute-value-expression cannot be a structured type (SQLSTATE 42884). The expression can be any SQL expression. If the expression is not a simple column reference, an attribute name must be specified.

**attribute-name**

Specifies an attribute name. The name is an SQL identifier that must be in the form of an XML qualified name, or QName (SQLSTATE 42634). See the [W3C XML namespace specifications](https://www.w3.org/TR/xml-names) for more details on valid names. The attribute name cannot be xmlns or prefixed with xmlns:. A namespace is declared using the function XMLNAMESPACES. Duplicate attribute names, whether implicit or explicit, are not allowed (SQLSTATE 42713).

If attribute-name is not specified, attribute-value-expression must be a column name (SQLSTATE 42703). The attribute name is created from the column name using the fully escaped mapping from a column name to an XML attribute name.

The data type of the result is XML. If the result of attribute-value-expression can be null, the result can be null; if the result of every attribute-value-expression is null, the result is the null value.

Examples:

**Note:** XMLATTRIBUTES does not insert blank spaces or new line characters in the output. All example output has been formatted to enhance readability.

- Produce an element with attributes.

  ```sql
  SELECT E.EMPNO, XMLELEMENT(
    NAME "Emp",
    XMLATTRIBUTES(
      E.EMPNO, E.FIRSTNAME || ' ' || E.LASTNAME AS "name"
    )
  ) AS "Result"
  FROM EMPLOYEE E WHERE E.EDLEVEL = 12
  ```

  This query produces the following result:
• Produce an element with a namespace declaration that is not used in any QName. The prefix is used in an attribute value.

VALUES XMLELEMENT(
    NAME "size",
    XMLNAMESPACES(
        'http://www.w3.org/2001/XMLSchema-instance' AS "xsi",
        'http://www.w3.org/2001/XMLSchema' AS "xsd"
    ),
    XMLATTRIBUTES(
        'xsd:string' AS "xsi:type"
    ),'1'
)

This query produces the following result:
<size xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      xsi:type="xsd:string">1</size>
The schema is SYSIBM. The function name cannot be specified as a qualified name.

The XMLCOMMENT function returns an XML value with a single XQuery comment node with the input argument as the content.

string-expression
An expression whose value has a character string type: CHAR, VARCHAR or CLOB. The result of the string-expression is parsed to check for conformance to the requirements for an XML comment, as specified in the XML 1.0 rule. The result of the string-expression must conform to the following regular expression:

\((\text{Char} \ - \ '-') \ | \ ('-' \ (\text{Char} \ - \ '-')))^*\)

where \text{Char} is defined as any Unicode character excluding surrogate blocks X’FFFE’ and X’FFFF’. Basically, the XML comment cannot contain two adjacent hyphens, and cannot end with a hyphen (SQLSTATE 2200S).

The data type of the result is XML. If the result of string-expression can be null, the result can be null; if the input value is null, the result is the null value.
XMLCONCAT

XMLCONCAT

\[ \text{XMLCONCAT}(\text{XML-expression}, \text{XML-expression}) \]

The schema is SYSIBM. The function name cannot be specified as a qualified name.

The XMLCONCAT function returns a sequence containing the concatenation of a variable number of XML input arguments.

XML-expression

- Specifies an expression of data type XML.

The data type of the result is XML. The result is an XML sequence containing the concatenation of the non-null input XML values. Null values in the input are ignored. If the result of any XML-expression can be null, the result can be null; if the result of every input value is null, the result is the null value.

Example:

Note: XMLCONCAT does not insert blank spaces or new line characters in the output. All example output has been formatted to enhance readability.

- Construct a department element for departments A00 and B01, containing a list of employees sorted by first name. Include an introductory comment immediately preceding the department element.

```sql
SELECT XMLCONCAT(
    XMLCOMMENT('Confirm these employees are on track for their product schedule'),
    XMLELEMENT(NAME "Department",
        XMLATTRIBUTES(E.WORKDEPT AS "name"),
    XMLAGG(
        XMLELEMENT(NAME "emp",
            E.FIRSTNME)
    ORDER BY E.FIRSTNME
    ))
)
FROM EMPLOYEE E
WHERE E.WORKDEPT IN ('A00', 'B01')
GROUP BY E.WORKDEPT
```

This query produces the following result:

```xml
<!--Confirm these employees are on track for their product schedule-->
<Department name="A00">
    <emp>CHRISTINE</emp>
    <emp>DIAN</emp>
    <emp>GREG</emp>
    <emp>SEAN</emp>
    <emp>VINCENZO</emp>
</Department>
<!--Confirm these employees are on track for their product schedule-->
<Department name="B01">
    <emp>MICHAEL</emp>
</Department>
```

Chapter 3. Functions 631
XMLDOCUMENT

The schema is SYSIBM. The function name cannot be specified as a qualified name.

The XMLDOCUMENT function returns an XML value with a single XQuery
document node with zero or more children nodes.

XML-expression

An expression that returns an XML value. A sequence item in the XML value
must not be an attribute node (SQLSTATE 10507).

The data type of the result is XML. If the result of XML-expression can be null, the
result can be null; if the input value is null, the result is the null value.

The children of the resulting document node are constructed as described in the
following steps. The input expression is a sequence of nodes or atomic values,
which is referred to in these steps as the content sequence.

1. If the content sequence contains a document node, the document node is
   replaced in the content sequence by the children of the document node.
2. Each adjacent sequence of one or more atomic values in the content sequence
   are replaced with a text node containing the result of casting each atomic value
to a string with a single blank character inserted between adjacent values.
3. For each node in the content sequence, a new deep copy of the node is
   constructed. A deep copy of a node is a copy of the whole subtree rooted at
   that node, including the node itself and its descendants. Each copied node has
   a new node identity.
4. The nodes in the content sequence become the children of the new document
   node.

The XMLDOCUMENT function effectively executes the XQuery computed
document constructor. The result of
XMLQUERY('document {$E}' PASSING BY REF XML-expression AS "E")

is equivalent to
XMLDOCUMENT( XML-expression )

with the exception of the case where XML-expression is null and XMLQUERY
returns the empty sequence compared to XMLDOCUMENT which returns the null
value.

Example:

- Insert a constructed document into an XML column.

  INSERT INTO T1 VALUES(
    123, (
      SELECT XMLDOCUMENT(
        XMLELEMENT( 
          NAME "Emp", E.FIRSTNAME || ' ' || E.LASTNAME, XMLCOMMENT('This is just a simple example')
        )
      )
    )
  )
XMLDOCUMENT

FROM EMPLOYEE E
WHERE E.EMPNO = '000120'
)
)

Chapter 3. Functions  633
**XML ELEMENT**

```
XMLELEMENT( NAME element-name ,
            xmlnamespaces-declaration ,
            xmlattributes-function ,
            element-content-expression )
```

Notes:
1. The OPTION clause can only be specified if at least one xmlattributes-function or element-content-expression is specified.
2. NULL ON NULL or EMPTY ON NULL can only be specified if at least one element-content-expression is specified.
3. The same clause must not be specified more than once.

The schema is SYSIBM. The function name cannot be specified as a qualified name.

The XMLELEMENT function returns an XML value that is an XQuery element node.

**NAME element-name**
- Specifies the name of an XML element. The name is an SQL identifier that must be in the form of an XML qualified name, or QName (SQLSTATE 42634). See the [W3C XML namespace specifications](#) for more details on valid names. If the name is qualified, the namespace prefix must be declared within the scope (SQLSTATE 42635).

**xmlnamespaces-declaration**
- Specifies the XML namespace declarations that are the result of the XMLNAMESPACES declaration. The namespaces that are declared are in the scope of the XMLELEMENT function. The namespaces apply to any nested XML functions within the XMLELEMENT function, regardless of whether or not they appear inside another subselect.

If xmlnamespaces-declaration is not specified, namespace declarations are not associated with the constructed element.

**xmlattributes-function**
- Specifies the XML attributes for the element. The attributes are the result of the XMLATTRIBUTES function.

**element-content-expression**
- The content of the generated XML element node is specified by an expression or a list of expressions. The data type of element-content-expression cannot be a structured type (SQLSTATE 42884). The expression can be any SQL expression.
XMLELEMENT

If element-content-expression is not specified, an empty string is used as the
content for the element and OPTION NULL ON NULL or EMPTY ON NULL
must not be specified.

OPTION
Specifies additional options for constructing the XML element. If no OPTION
clause is specified, the default is EMPTY ON NULL XMLBINARY USING
BASE64. This clause has no impact on nested XMLELEMENT invocations
specified in element-content-expression.

EMPTY ON NULL or NULL ON NULL
Specifies whether a null value or an empty element is to be returned if the
values of each element-content-expression is a null value. This option only
affects null handling of element contents, not attribute values. The default
is EMPTY ON NULL.

EMPTY ON NULL
If the value of each element-content-expression is null, an empty element
is returned.

NULL ON NULL
If the value of each element-content-expression is null, a null value is
returned.

XMLBINARY USING BASE64 or XMLBINARY USING HEX
Specifies the assumed encoding of binary input data, character string data
with the FOR BIT DATA attribute, or a distinct type that is based on one of
these types. The encoding applies to element content or attribute values.
The default is XMLBINARY USING BASE64.

XMLBINARY USING BASE64
Specifies that the assumed encoding is base64 characters, as defined for
XML schema type xs:base64Binary encoding. The base64 encoding uses
a 65-character subset of US-ASCII (10 digits, 26 lowercase characters,
26 uppercase characters, ‘+’, and ‘/’) to represent every six bits of the
binary or bit data with one printable character in the subset. These
characters are selected so that they are universally representable. Using
this method, the size of the encoded data is 33 percent larger than the
original binary or bit data.

XMLBINARY USING HEX
Specifies that the assumed encoding is hexadecimal characters, as
defined for XML schema type xs:hexBinary encoding. The hexadecimal
encoding represents each byte (8 bits) with two hexadecimal characters.
Using this method, the encoded data is twice the size of the original
binary or bit data.

This function takes an element name, an optional collection of namespace
declarations, an optional collection of attributes, and zero or more arguments that
make up the content of the XML element. The result is an XML sequence
containing an XML element node or the null value.

The data type of the result is XML. If any of the element-content-expression
arguments can be null, the result can be null; if all the element-content-expression
argument values are null and the NULL ON NULL option is in effect, the result is
the null value.

Note:
1. When constructing elements that will be copied as content of another element
   that defines default namespaces, default namespaces should be explicitly
   undeclared in the copied element to avoid possible errors that could result from
   inheriting the default namespace from the new parent element. Predefined
   namespace prefixes (‘xs’, ‘xsi’, ‘xml’, and ‘sqlxml’) must also be declared
   explicitly when they are used.

2. **Constructing an element node:** The resulting element node is constructed as
   follows:
   
a. The `xmlnamespaces-declaration` adds a set of in-scope namespaces for the
      constructed element. Each in-scope namespace associates a namespace
      prefix (or the default namespace) with a namespace URI. The in-scope
      namespaces define the set of namespace prefixes that are available for
      interpreting QNames within the scope of the element.
   
b. If the `xmlattributes-function` is specified, it is evaluated and the result is a
      sequence of attribute nodes.
   
c. Each `element-content-expression` is evaluated and the result is converted into a
      sequence of nodes as follows:
      
      - If the result type is not XML, it is converted to an XML text node whose
        content is the result of `element-content-expression` mapped to XML
        according to the rules of mapping SQL data values to XML data values
        (see the table that describes supported casts from non-XML values to
        XML values in “Casting between data types”).
      
      - If the result type is XML, then in general the result is a sequence of items.
        Some of the items in that sequence might be document nodes. Each
        document node in the sequence is replaced by the sequence of its
        top-level children. Then for each node in the resulting sequence, a new
        deep copy of the node is constructed, including its children and
        attributes. Each copied node has a new node identity. Copied element
        and attribute nodes preserve their type annotation. For each adjacent
        sequence of one or more atomic values returned in the sequence, a new
        text node is constructed, containing the result of casting each atomic
        value to a string, with a single blank character inserted between adjacent
        values. Adjacent text nodes in the content sequence are merged into a
        single text node by concatenating their contents, with no intervening
        blanks. After concatenation, any text node whose content is a zero-length
        string is deleted from the content sequence.

   d. The result sequence of XML attributes and the resulting sequences of all
      `element-content-expression` specifications are concatenated into one sequence
      which is called the content sequence. Any sequence of adjacent text nodes
      in the content sequence is merged into a single text node. If all the
      `element-content-expression` arguments are empty strings, or an
      `element-content-expression` argument is not specified, an empty element is
      returned.

   e. The content sequence must not contain an attribute node following a node
      that is not an attribute node (SQLSTATE 10507). Attribute nodes occurring
      in the content sequence become attributes of the new element node. Two or
      more of these attribute nodes must not have the same name (SQLSTATE
      10503). A namespace declaration is created corresponding to any namespace
      used in the names of the attribute nodes if the namespace URI is not in the
      in-scope namespaces of the constructed element.

   f. Element, text, comment, and processing instruction nodes in the content
      sequence become the children of the constructed element node.
The constructed element node is given a type annotation of `xs:anyType`, and each of its attributes is given a type annotation of `xdt:untypedAtomic`. The node name of the constructed element node is `element-name` specified after the `NAME` keyword.

3. **Rules for using namespaces within `XMLELEMENT`**: Consider the following rules about scoping of namespaces:

- The namespaces declared in the `XMLNAMESPACES` declaration are the in-scope namespaces of the element node constructed by the `XMLELEMENT` function. If the element node is serialized, then each of its in-scope namespaces will be serialized as a namespace attribute unless it is an in-scope namespace of the parent of the element node and the parent element is serialized too.

- If an `XMLQUERY` or `XMLEXISTS` is in an `element-content-expression`, then the namespaces becomes the statically known namespaces of the XQuery expression of the `XMLQUERY` or `XMLEXISTS`. Statically known namespaces are used to resolve the QNames in the XQuery expression. If the XQuery prolog declares a namespace with the same prefix, within the scope of the XQuery expression, the namespace declared in the prolog will override the namespaces declared in the `XMLNAMESPACES` declaration.

- If an attribute of the constructed element comes from an `element-content-expression`, its namespace might not already be declared as an in-scope namespace of the constructed element, in this case, a new namespace is created for it. If this would result in a conflict, which means that the prefix of the attribute name is already bound to a different URI by a in-scope namespace, DB2 generates a prefix that does not cause such a conflict and the prefix used in the attribute name is changed to the new prefix, and a namespace is created for this new prefix. The generated new prefix follows the following pattern: "db2ns-xx", where "x" is a character chosen from the set [A-Z,a-z,0-9]. For example:

```sql
VALUES XMLELEMENT
  NAME "c", XMLQUERY(
    'declare namespace ipo="www.ipo.com"; $m/ipo:a/@ipo:b'
    PASSING XMLPARSE(
      DOCUMENT '<tst:a xmlns:tst="www.ipo.com" tst:b="2"/>'
    ) AS "m"
  )

returns:
<c xmlns:tst="www.ipo.com" tst:b="2"/>
```

A second example:

```sql
VALUES XMLELEMENT(
  NAME "tst:c", XMLNAMESPACES(
    'www.tst.com' AS "tst"
  ),
  XMLQUERY(
    'declare namespace ipo="www.ipo.com"; $m/ipo:a/@ipo:b'
    PASSING XMLPARSE(
      DOCUMENT '<tst:a xmlns:tst="www.ipo.com" tst:b="2"/>'
    ) AS "m"
  )
)

returns:
<tst:c xmlns:tst="www.tst.com" xmlns:db2ns-a1="www.ipo.com"
    db2ns-a1:b="2"/>
```
**XMLELEMENT**

Examples:

Note: XMLELEMENT does not insert blank spaces or new line characters in the output. All example output has been formatted to enhance readability.

- Construct an element with the NULL ON NULL option.

```sql
SELECT E.FIRSTNME, E.LASTNAME,
XMLELEMENT(
    NAME "Emp",
    XMLELEMENT(
        NAME "firstname", E.FIRSTNME
    ),
    XMLELEMENT(
        NAME "lastname", E.LASTNAME
    )
    OPTION NULL ON NULL
) AS "Result"
FROM EMPLOYEE E
WHERE E.EDLEVEL = 12
```

This query produces the following result:

<table>
<thead>
<tr>
<th>FIRSTNME</th>
<th>LASTNAME</th>
<th>Emp</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>JOHN</td>
<td>PARKER</td>
<td>&lt;Emp&gt;</td>
<td>&lt;emp&gt;JOHN&lt;/emp&gt;</td>
</tr>
<tr>
<td>MAUDE</td>
<td>SETRIGHT</td>
<td>&lt;Emp&gt;</td>
<td>&lt;emp&gt;MAUDE&lt;/emp&gt;</td>
</tr>
<tr>
<td>MICHELLE</td>
<td>SPRINGER</td>
<td>&lt;Emp&gt;</td>
<td>&lt;emp&gt;MICHELLE&lt;/emp&gt;</td>
</tr>
</tbody>
</table>

- Produce an element with a list of elements nested as child elements.

```sql
SELECT XMLELEMENT(
    NAME "Department",
    XMLATTRIBUTES(
        E.WORKDEPT AS "name"
    ),
    XMLAGG(
        XMLELEMENT(
            NAME "emp", E.FIRSTNME
        )
        ORDER BY E.FIRSTNME
    )
) AS "dept_list"
FROM EMPLOYEE E
WHERE E.WORKDEPT IN ('A00', 'B01')
GROUP BY WORKDEPT
```

This query produces the following result:

```
dept_list
<Department name="A00">
<emp>CHRISTINE</emp>
<emp>SEAN</emp>
<emp>VINCENZO</emp>
</Department>
<Department name="B01">
<emp>MICHAEL</emp>
</Department>
```

- Creating nested XML elements specifying a default XML element namespace and using a subselect.

```sql
SELECT XMLELEMENT(
    NAME "root",
    XMLNAMESPACES(DEFAULT 'http://mytest.uri'),
    XMLATTRIBUTES(cid),
    (SELECT
        XMLAGG(
            XMLELEMENT(
```
The statement returns the following XML document with the default element namespace declared in the root element:

```xml
<root xmlns="http://mytest.uri" CID="1002">
  <poid>5000</poid>
  <poid>5003</poid>
  <poid>5006</poid>
</root>
```

Using a common table expression with XML namespaces.

When an XML element is constructed with a common table expression and the element is used in elsewhere in the same SQL statement, any namespace declarations should be specified as part of the element construction. The following statement specifies the default XML namespace in both the common table expression that uses the PURCHASEORDER table to create the poid elements and the SELECT statement that uses the CUSTOMER table to create the root element.

```sql
WITH tempid(id, elem) AS
  (SELECT custid,
       XMLELEMENT(NAME "poid",
                  XMLNAMESPACES(DEFAULT 'http://mytest.uri'),
                  poid)
       FROM purchaseorder )
SELECT
       XMLELEMENT(NAME "root",
                  XMLNAMESPACES(DEFAULT 'http://mytest.uri'),
                  XMLATTRIBUTES(cid),
                  (SELECT XMLAGG(elem)
                    FROM tempid
                    WHERE tempid.id = customer.cid )
  FROM customer
WHERE cid = '1002'
```

The statement returns the following XML document with a default element namespace declared in the root element.

```xml
<root xmlns="http://mytest.uri" CID="1002">
  <poid>5000</poid>
  <poid>5003</poid>
  <poid>5006</poid>
</root>
```

In the following statement, the default element namespace is declared only in the SELECT statement that uses the CUSTOMER table to create the root element:

```sql
WITH tempid(id, elem) AS
  (SELECT custid, XMLELEMENT(NAME "poid", poid)
   FROM purchaseorder )
SELECT XMLELEMENT(NAME "root",
                   XMLNAMESPACES(DEFAULT 'http://mytest.uri'),
                   XMLATTRIBUTES(cid),
                   (SELECT XMLAGG(elem)
                    FROM tempid
                    WHERE tempid.id = customer.cid )
  FROM customer
WHERE cid = '1002'
```
The statement returns the following XML document with the default element namespace declared in the root element. Because the poid elements are created in the common table expression without a default element namespace declaration, the default element namespace for the poid elements is not defined. In the XML document, the default element namespace for the poid elements is set to an empty string "" because the default element namespace for the poid elements is not defined, and the poid elements do not belong to the default element namespace of the root element xmlns="http://mytest.uri".

```xml
<root xmlns="http://mytest.uri" CID="1002">
  <poid xmlns="">5000</poid>
  <poid xmlns="">5003</poid>
  <poid xmlns="">5006</poid>
</root>
```
Notes:
1. The same clause must not be specified more than once.

The schema is SYSIBM. The function name cannot be specified as a qualified name.

The XMLFOREST function returns an XML value that is a sequence of XQuery element nodes.

xmlnamespaces-declaration
   Specifies the XML namespace declarations that are the result of the XMLNAMESPACES declaration. The namespaces that are declared are in the scope of the XMLFOREST function. The namespaces apply to any nested XML functions within the XMLFOREST function, regardless of whether or not they appear inside another subselect.

   If xmlnamespaces-declaration is not specified, namespace declarations are not associated with the constructed elements.

element-content-expression
   The content of the generated XML element node is specified by an expression. The data type of element-content-expression cannot be a structured type (SQLSTATE 42884). The expression can be any SQL expression. If the expression is not a simple column reference, an element name must be specified.

AS element-name
   Specifies the XML element name as an SQL identifier. The element name must be of the form of an XML qualified name, or QName (SQLSTATE 42634). See the W3C XML namespace specifications for more details on valid names. If the name is qualified, the namespace prefix must be declared within the scope (SQLSTATE 42635). If element-name is not specified, element-content-expression must be a column name (SQLSTATE 42703). The element name is created from the column name using the fully escaped mapping from a column name to an QName.
OPTION
Specifies additional options for constructing the XML element. If no OPTION clause is specified, the default is NULL ON NULL XMLBINARY USING BASE64. This clause has no impact on nested XMLELEMENT invocations specified in element-content-expression.

EMPTY ON NULL or NULL ON NULL
Specifies whether a null value or an empty element is to be returned if the values of each element-content-expression is a null value. This option only affects null handling of element contents, not attribute values. The default is NULL ON NULL.

EMPTY ON NULL
If the value of each element-content-expression is null, an empty element is returned.

NULL ON NULL
If the value of each element-content-expression is null, a null value is returned.

XMLBINARY USING BASE64 or XMLBINARY USING HEX
Specifies the assumed encoding of binary input data, character string data with the FOR BIT DATA attribute, or a distinct type that is based on one of these types. The encoding applies to element content or attribute values. The default is XMLBINARY USING BASE64.

XMLBINARY USING BASE64
Specifies that the assumed encoding is base64 characters, as defined for XML schema type xs:base64Binary encoding. The base64 encoding uses a 65-character subset of US-ASCII (10 digits, 26 lowercase characters, 26 uppercase characters, ‘+’, and ‘/’) to represent every six bits of the binary or bit data with one printable character in the subset. These characters are selected so that they are universally representable. Using this method, the size of the encoded data is 33 percent larger than the original binary or bit data.

XMLBINARY USING HEX
Specifies that the assumed encoding is hexadecimal characters, as defined for XML schema type xs:hexBinary encoding. The hexadecimal encoding represents each byte (8 bits) with two hexadecimal characters. Using this method, the encoded data is twice the size of the original binary or bit data.

This function takes an optional set of namespace declarations and one or more arguments that make up the name and element content for one or more element nodes. The result is an XML sequence containing a sequence of XQuery element nodes or the null value.

The data type of the result is XML. If any of the element-content-expression arguments can be null, the result can be null; if all the element-content-expression argument values are null and the NULL ON NULL option is in effect, the result is the null value.

The XMLFOREST function can be expressed by using XMLCONCAT and XMLELEMENT. For example, the following two expressions are semantically equivalent.

```
XMLFOREST(xmlnamespaces-declaration, arg1 AS name1, arg2 AS name2 ...)
```
Note:
1. When constructing elements that will be copied as content of another element that defines default namespaces, default namespaces should be explicitly undeclared in the copied element to avoid possible errors that could result from inheriting the default namespace from the new parent element. Predefined namespace prefixes ('xs', 'xsi', 'xml', and 'sqlxml') must also be declared explicitly when they are used.

Example:

Note: XMLFOREST does not insert blank spaces or new line characters in the output. All example output has been formatted to enhance readability.

- Construct a forest of elements with a default namespace.

```sql
SELECT EMPNO,
XMLFOREST(  
XMLNAMESPACES(  
   DEFAULT 'http://hr.org', 'http://fed.gov' AS "d"  
  ),
  LASTNAME, JOB AS "d:job"
) AS "Result"
FROM EMPLOYEE
WHERE EDLEVEL = 12
```

This query produces the following result:

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>Result</th>
</tr>
</thead>
</table>
| 000290 | <LASTNAME xmlns="http://hr.org" xmlns:d="http://fed.gov">PARKER</LASTNAME>
| 000310 | <LASTNAME xmlns="http://hr.org" xmlns:d="http://fed.gov">SETRIGHT</LASTNAME>
| 200310 | <LASTNAME xmlns="http://hr.org" xmlns:d="http://fed.gov">SPRINGER</LASTNAME>
XMLNAMESPACES

XMLNAMESPACES

xmlnamespaces-declaration:

```
XMLNAMESPACES ( ( namespace-uri AS namespace-prefix ) )
```

Notes:

1. DEFAULT or NO DEFAULT can only be specified once in arguments of XMLNAMESPACES.

The schema is SYSIBM. The declaration name cannot be specified as a qualified name.

The XMLNAMESPACES declaration constructs namespace declarations from the arguments. This declaration can only be used as an argument for specific functions such as XMLELEMENT, XMLFOREST and XMTABLE. The result is one or more XML namespace declarations containing in-scope namespaces for each non-null input value.

namespace-uri

- Specifies the namespace universal resource identifier (URI) as an SQL character string constant. This character string constant must not be empty if it is used with a namespace-prefix (SQLSTATE 42815).

namespace-prefix

- Specifies a namespace prefix. The prefix is an SQL identifier that must be in the form of an XML NCName (SQLSTATE 42634). See the W3C XML namespace specifications for more details on valid names. The prefix cannot be xml or xmlns and the prefix must be unique within the list of namespace declarations (SQLSTATE 42635).

DEFAULT namespace-uri

- Specifies the default namespace to use within the scope of this namespace declaration. The namespace-uri applies for unqualified names in the scope unless overridden in a nested scope by another DEFAULT declaration or a NO DEFAULT declaration.

NO DEFAULT

- Specifies that no default namespace is to be used within the scope of this namespace declaration. There is no default namespace in the scope unless overridden in a nested scope by a DEFAULT declaration.

The data type of the result is XML. The result is an XML namespace declaration for each specified namespace. The result cannot be null.

Examples:

- Produce an XML element named adm:employee and an XML attribute adm:department, both associated with a namespace whose prefix is adm.

Note: XMLNAMESPACES does not insert blank spaces or new line characters in the output. All example output has been formatted to enhance readability.
XMLNAMESPACES

```
SELECT EMPNO, XMLELEMENT(
    NAME "adm:employee", XMLNAMESPACES(
        'http://www.adm.com' AS "adm"
    ),
    XMLATTRIBUTES(
        WORKDEPT AS "adm:department"
    ),
    LASTNAME
) FROM EMPLOYEE
WHERE JOB = 'ANALYST'
```

This query produces the following result:

```
000130 <adm:employee xmlns:adm="http://www.adm.com" adm:department="C01">
    QUINTANA
</adm:employee>
000140 <adm:employee xmlns:adm="http://www.adm.com" adm:department="C01">
    NICHOLLS
</adm:employee>
200140 <adm:employee xmlns:adm="http://www.adm.com" adm:department="C01">
    NATZ
</adm:employee>
```

- Produce an XML element named 'employee', which is associated with a default namespace, and a sub-element named 'job', which does not use a default namespace, but whose sub-element named 'department' does use a default namespace.

```
SELECT EMP.EMPNO, XMLELEMENT(
    NAME "employee", XMLNAMESPACES(
        DEFAULT 'http://hr.org'
    ),
    EMP.LASTNAME, XMLELEMENT(
        NAME "job", XMLNAMESPACES(
            NO DEFAULT
        ),
        EMP.JOB, XMLELEMENT(
            NAME "department", XMLNAMESPACES(
                DEFAULT 'http://adm.org'
            ),
            EMP.WORKDEPT
        )
    )
)
FROM EMPLOYEE EMP
WHERE EMP.EDLEVEL = 12
```

This query produces the following result:

```
000290 <employee xmlns="http://hr.org">PARKER<job xmlns="">OPERATOR
    <department xmlns="http://hr.org">E11</department></job></employee>
000310 <employee xmlns="http://hr.org">SETRIGHT<job xmlns="">OPERATOR
    <department xmlns="http://hr.org">E11</department></job></employee>
200310 <employee xmlns="http://hr.org">SPRINGER<job xmlns="">OPERATOR
    <department xmlns="http://hr.org">E11</department></job></employee>
```
The schema is SYSIBM. The function name cannot be specified as a qualified name.

The XMLPARSE function parses the argument as an XML document and returns an XML value.

**DOCUMENT**
Specify that the character string expression to be parsed must evaluate to a well-formed XML document that conforms to XML 1.0, as modified by the XML Namespaces recommendation (SQLSTATE 2200M).

**string-expression**
Specify an expression that returns a character string or BLOB value. If a parameter marker is used, it must explicitly be cast to one of the supported data types.

**STRIP WHITESPACE or PRESERVE WHITESPACE**
Specify whether or not whitespace in the input argument is to be preserved. If neither is specified, STRIP WHITESPACE is the default.

**STRIP WHITESPACE**
Specify that text nodes containing only whitespace characters up to 1000 bytes in length will be stripped, unless the nearest containing element has the attribute xml:space='preserve'. If any text node begins with more than 1000 bytes of whitespace, an error is returned (SQLSTATE 54059).

The whitespace characters in the CDATA section are also affected by this option. DTDs may have DOCTYPE declarations for elements, but the content models of elements are not used to determine if whitespace is stripped or not.

**PRESERVE WHITESPACE**
Specify that all whitespace is to be preserved, even when the nearest containing element has the attribute xml:space='default'.

The data type of the result is XML. If the result of **string-expression** can be null, the result can be null; if the result of **string-expression** is null, the result is the null value.

**Note:**
1. **Encoding of the input string:** The input string may contain an XML declaration that identifies the encoding of the characters in the XML document. If the string is passed to the XMLPARSE function as a character string, it will be converted to the code page at the database server. This code page may be different from the originating code page and the encoding identified in the XML declaration.

   Therefore, applications should avoid direct use of XMLPARSE with character string input and should send strings containing XML documents directly using host variables to maintain the match between the external code page and the encoding in the XML declaration. If XMLPARSE must be used in this situation, a BLOB type should be specified as the argument to avoid code page conversion.
2. **Handling of DTDs**: External document type definitions (DTDs) and entities must be registered in a database. Both internal and external DTDs are checked for valid syntax. During the parsing process, the following actions are also performed:

- Default values that are defined by the internal and external DTDs are applied.
- Entity references and parameter entities are replaced by their expanded forms.
- If an internal DTD and an external DTD define the same element, an error is returned (SQLSTATE 2200M).
- If an internal DTD and an external DTD define the same entity or attribute, the internal definition is chosen.

After parsing, internal DTDs and entities, as well as references to external DTDs and entities, are not preserved in the stored representation of the value.

3. **Character conversion in non-UTF-8 databases**: Code page conversion occurs when an XML document is parsed into a non-Unicode database server, if the document is passed in from a host variable or parameter marker of a character data type, or from a character string literal. Parsing an XML document using a host variable or parameter marker of type XML, BLOB or FOR BIT DATA (CHAR FOR BIT DATA or VARCHAR FOR BIT DATA) prevents code page conversion. When a character data type is used, care must be taken to ensure that all characters in the XML document have a matching code point in the target database code page, otherwise substitution characters may be introduced. The configuration parameter `enable_xmlchar` can be used to help ensure the integrity of XML data stored in a non-Unicode database. Setting this parameter to "NO" blocks the insertion of XML documents from character data types. The BLOB and FOR BIT DATA data types are still allowed, as documents passed into a database using these data types avoid code page conversion.

**Example**

Using the PRESERVE WHITESPACE option preserves the white space characters in the XML document inserted into the table, including the white space characters in the description element.

```sql
INSERT INTO PRODUCT VALUES ('100-103-99','Tool bag',14.95,NULL,NULL,NULL,
XMLPARSE(DOCUMENT
  '<produce xmlns="http://posample.org" pid="100-103-99">
    <description>
      <name>Tool bag</name>
      <details>
        Super Deluxe tool bag:
        - 26 inches long, 12 inches wide
        - Curved padded handle
        - Locking latch
        - Reinforced exterior pockets
      </details>
    </description>
    <price>14.95</price>
    <weight>3 kg</weight>
  </produce>')
PRESERVE WHITESPACE ));
```

Running the following select statement

```sql
SELECT XMLQUERY ('$/*:product/*:description/*:details' PASSING DESCRIPTION as "d")
FROM PRODUCT WHERE PID = '100-103-99' ;
```

returns the details element with the white space characters:
<details xmlns="http://posample.org">
  Super Deluxe tool bag:
  - 26 inches long, 12 inches wide
  - Curved padded handle
  - Locking latch
  - Reinforced exterior pockets
</details>
XMLPI

The schema is SYSIBM. The function name cannot be specified as a qualified name.

The XMLPI function returns an XML value with a single XQuery processing instruction node.

**NAME pi-name**

Specifies the name of a processing instruction. The name is an SQL identifier that must be in the form of an XML NCName (SQLSTATE 42634). See the [W3C XML namespace specifications](https://www.w3.org/TR/REC-xml-names) for more details on valid names. The name cannot be the word "xml" in any case combination (SQLSTATE 42634).

**string-expression**

An expression that returns a value that is a character string. The resulting string is converted to UTF-8 and must conform to the content of an XML processing instruction as specified in XML 1.0 rules (SQLSTATE 2200T):

- The string must not contain the substring `?>` since this substring terminates a processing instruction
- Each character of the string can be any Unicode character excluding the surrogate blocks, X'FFFE' and X'FFFF'.

The resulting string becomes the content of the constructed processing instruction node.

The data type of the result is XML. If the result of `string-expression` can be null, the result can be null; if the result of `string-expression` is null, the result is the null value. If `string-expression` is an empty string or is not specified, an empty processing instruction node is returned.

Examples:

- Generate an XML processing instruction node.

  ```sql
  SELECT XMLPI(  
    NAME "Instruction", 'Push the red button'
  )
  FROM SYSIBM.SYSDUMMY1
  ```

  This query produces the following result:
  ```xml
  <?Instruction Push the red button?>
  ```

- Generate an empty XML processing instruction node.

  ```sql
  SELECT XMLPI(  
    NAME "Warning"
  )
  FROM SYSIBM.SYSDUMMY1
  ```

  This query produces the following result:
  ```xml
  <?Warning ?>
  ```
The XMLQUERY function returns an XML value from the evaluation of an XQuery expression possibly using specified input arguments as XQuery variables.

**xquery-expression-constant**

Specifies an SQL character string constant that is interpreted as an XQuery expression using supported XQuery language syntax. The constant string is converted to UTF-8 before being parsed as an XQuery statement. The XQuery expression executes using an optional set of input XML values, and returns an output sequence that is also returned as the value of the XMLQUERY expression. The value for `xquery-expression-constant` must not be an empty string or a string of blank characters (SQLSTATE 10505).

**PASSING**

Specifies input values and the manner in which these values are passed to the XQuery expression specified by `xquery-expression-constant`. By default, every unique column name that is in the scope where the function is invoked is implicitly passed to the XQuery expression using the name of the column as the variable name. If an `identifier` in a specified `xquery-argument` matches an in-scope column name, then the explicit `xquery-argument` is passed to the XQuery expression overriding that implicit column.

**BY REF**

Specifies that the default passing mechanism is by reference for any `xquery-variable-expression` of data type XML and for the returned value. When XML values are passed by reference, the XQuery evaluation uses the input node trees, if any, directly from the specified input expressions, preserving all properties, including the original node identities and document order. If two arguments pass the same XML value, node identity

Notes:

1. The data type of the expression cannot be DECFLOAT.

The schema is SYSIBM. The function name cannot be specified as a qualified name.
comparisons and document ordering comparisons involving some nodes contained between the two input arguments might refer to nodes within the same XML node tree.

This clause has no impact on how non-XML values are passed. The non-XML values create a new copy of the value during the cast to XML.

**xquery-argument**

Specifies an argument that is to be passed to the XQuery expression specified by `xquery-expression-constant`. An argument specifies a value and the manner in which that value is to be passed. The argument includes an SQL expression that is evaluated.

- If the resulting value is of type XML, it becomes an `input-xml-value`. A null XML value is converted to an XML empty sequence.
- If the resulting value is not of type XML, it must be castable to the XML data type. A null value is converted to an XML empty sequence. The converted value becomes an `input-xml-value`.

When the `xquery-expression-constant` is evaluated, an XQuery variable is presented with a value equal to `input-xml-value` and a name specified by the AS clause.

**xquery-variable-expression**

Specifies an SQL expression whose value is available to the XQuery expression specified by `xquery-expression-constant` during execution. The expression cannot contain a sequence reference (SQLSTATE 428F9) or an OLAP function (SQLSTATE 42903). The data type of the expression cannot be DECFLOAT.

**AS identifier**

Specifies that the value generated by `xquery-variable-expression` will be passed to `xquery-expression-constant` as an XQuery variable. The variable name will be `identifier`. The leading dollar sign ($) that precedes variable names in the XQuery language is not included in `identifier`. The identifier must be a valid XQuery variable name and is restricted to an XML NCName (SQLSTATE 42634). The identifier must not be greater than 128 bytes in length. Two arguments within the same PASSING clause cannot use the same identifier (SQLSTATE 42711).

**BY REF**

Indicates that an XML input value is to be passed by reference. When XML values are passed by reference, the XQuery evaluation uses the input node trees, if any, directly from the specified input expressions, preserving all properties, including the original node identities and document order. If two arguments pass the same XML value, node identity comparisons and document ordering comparisons involving some nodes contained between the two input arguments might refer to nodes within the same XML node tree. If BY REF is not specified following an `xquery-variable-expression`, XML arguments are passed by way of the default passing mechanism that is provided through the syntax that follows the PASSING keyword. This option cannot be specified for non-XML values. When a non-XML value is passed, the value is converted to XML; this process creates a copy.

**RETURNING SEQUENCE**

Indicates that the XMLQUERY expression returns a sequence.

**BY REF**

Indicates that the result of the XQuery expression is returned by reference. If
XMLQUERY

this value contains nodes, any expression using the return value of the XQuery expression will receive node references directly, preserving all node properties, including the original node identities and document order. Referenced nodes will remain connected within their node trees. If the BY REF clause is not specified and the PASSING is specified, the default passing mechanism is used. If BY REF is not specified and PASSING is not specified, the default returning mechanism is BY REF.

EMPTY ON EMPTY

Specifies that an empty sequence result from processing the XQuery expression is returned as an empty sequence.

The data type of the result is XML; it cannot be null.

If the evaluation of the XQuery expression results in an error, then the XMLQUERY function returns the XQuery error (SQLSTATE class ‘10’).

Note:

1. XMLQUERY usage restrictions: The XMLQUERY function cannot be:
   • Part of the ON clause that is associated with a JOIN operator or a MERGE statement (SQLSTATE 42972)
   • Part of the GENERATE KEY USING or RANGE THROUGH clause in the CREATE INDEX EXTENSION statement (SQLSTATE 428E3)
   • Part of the FILTER USING clause in the CREATE FUNCTION (External Scalar) statement, or the FILTER USING clause in the CREATE INDEX EXTENSION statement (SQLSTATE 428E4)
   • Part of a check constraint or a column generation expression (SQLSTATE 42621)
   • Part of a group-by-clause (SQLSTATE 42822)
   • Part of an argument for a column-function (SQLSTATE 42607)

2. XMLQUERY as a subquery: An XMLQUERY expression that acts as a subquery can be restricted by statements that restrict subqueries.
XMLROW

The XMLROW function returns an XML value with a single XQuery document node containing one top-level element node.

```
XMLROW(expression AS qname-identifier)
```

Notes:
1. The same clause must not be specified more than once.

The schema is SYSIBM. The function name cannot be specified as a qualified name.

expression
   The content of each generated XML element node is specified by an expression.
   The data type of the expression cannot be a structured type (SQLSTATE 42884).
   The expression can be any SQL expression. If the expression is not a simple column reference, an element name must be specified.

AS qname-identifier
   Specifies the XML element name or attribute name as an SQL identifier. The qname-identifier must be of the form of an XML qualified name, or QName (SQLSTATE 42634). See the W3C XML namespace specifications for more details on valid names. If the name is qualified, the namespace prefix must be declared within the scope (SQLSTATE 42635). If qname-identifier is not specified, expression must be a column name (SQLSTATE 42703). The element name or attribute name is created from the column name using the fully escaped mapping from a column name to an QName.

OPTION
   Specifies additional options for constructing the XML value. If no OPTION clause is specified, the default behavior applies.

AS ATTRIBUTES
   Specifies that each expression is mapped to an attribute value with column name or qname-identifier serving as the attribute name.

ROW row-name
   Specifies the name of the element to which each row is mapped. If this option is not specified, the default element name is "row".

Notes

By default, each row in the result set is mapped to an XML value as follows:
- Each row is transformed into an XML element named "row" and each column is transformed into a nested element with the column name as the element name.
XMLROW

- The null handling behavior is NULL ON NULL. A NULL value in a column maps to the absence of the subelement. If all column values are NULL, a NULL value is returned by the function.
- The binary encoding scheme for BLOB and FOR BIT DATA data types is base64Binary encoding.
- A document node will be added implicitly to the row element to make the XML result a well-formed single-rooted XML document.

Examples

Assume the following table T1 with columns C1 and C2 that contain numeric data stored in a relational format:

<table>
<thead>
<tr>
<th>C1</th>
<th>C2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>-</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

4 record(s) selected.

- The following example shows an XMLRow query and output fragment with default behavior, using a sequence of row elements to represent the table:

```sql
SELECT XMLROW(C1, C2) FROM T1
<row><C1>1</C1><C2>2</C2></row>
<row><C2>2</C2></row>
<row><C1>1</C1></row>

4 record(s) selected.
```

- The following example shows an XMLRow query and output fragment with attribute centric mapping. Instead of appearing as nested elements as in the previous example, relational data is mapped to element attributes:

```sql
SELECT XMLROW(C1, C2 OPTION AS ATTRIBUTES) FROM T1
<row C1="1" C2="2"/>
<row C2="2"/>
<row C1="1"/>

4 record(s) selected.
```

- The following example shows an XMLRow query and output fragment with the default <row> element replaced by <entry>. Columns C1 and C2 are returned as <column1> and <column2> elements, and the total of C1 and C2 is returned inside a <total> element:

```sql
SELECT XMLROW(
    C1 AS "column1", C2 AS "column2",
    C1+C2 AS "total" OPTION ROW "entry")
FROM T1
<entry><column1>1</column1><column2>2</column2><total>3</total></entry>
<entry><column2>2</column2></entry>
<entry><column1>1</column1></entry>

4 record(s) selected.
XMLSERIALIZE

XMLSERIALIZE (XML-expression AS data-type 

  VERSION '1.0' 
  EXCLUDING XMLDECLARATION 
  INCLUDING XMLDECLARATION 

data-type:

<table>
<thead>
<tr>
<th>data-type</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHARACTER</td>
<td>(integer)</td>
</tr>
<tr>
<td>CHAR</td>
<td>(-integer-)</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>(integer)</td>
</tr>
<tr>
<td>CHAR</td>
<td>(-integer-)</td>
</tr>
<tr>
<td>CHARACTER VARYING</td>
<td>(integer)</td>
</tr>
<tr>
<td>CHAR</td>
<td>(-integer-)</td>
</tr>
<tr>
<td>CLOB</td>
<td>(1M)</td>
</tr>
<tr>
<td>CHARACTER LARGE OBJECT</td>
<td>(integer K M G)</td>
</tr>
<tr>
<td>CHAR</td>
<td>(-integer-)</td>
</tr>
<tr>
<td>BLOB</td>
<td>(1M)</td>
</tr>
<tr>
<td>BINARY LARGE OBJECT</td>
<td>(-integer K M G)</td>
</tr>
</tbody>
</table>

Notes:
1. The same clause must not be specified more than once.

The schema is SYSIBM. The function name cannot be specified as a qualified name.

The XMLSERIALIZE function returns a serialized XML value of the specified data type generated from the XML-expression argument.

CONTENT
  Specifies that any XML value can be specified and the result of the serialization is based on this input value.

XML-expression
  Specifies an expression that returns a value of data type XML. The XML sequence value must not contain an item that is an attribute node (SQLSTATE 2200W). This is the input to the serialization process.

AS data-type
  Specifies the result type. The implicit or explicit length attribute of the specified result data type must be sufficient to contain the serialized output (SQLSTATE 22001).

VERSION '1.0'
  Specifies the XML version of the serialized value. The only version supported is '1.0' which must be specified as a string constant (SQLSTATE 42815).
EXCLUDING XMLDECLARATION or INCLUDING XMLDECLARATION

Specifies whether an XML declaration is included in the result. The default is EXCLUDING XMLDECLARATION.

EXCLUDING XMLDECLARATION

Specifies that an XML declaration is not included in the result.

INCLUDING XMLDECLARATION

Specifies that an XML declaration is included in the result. The XML declaration is the string `<?xml version="1.0" encoding="UTF-8"?>`.

The result has the data type specified by the user. An XML sequence is effectively converted to have a single document node by applying XMLDOCUMENT to XML-expression prior to serializing the resulting XML nodes. If the result of XML-expression can be null, the result can be null; if the result of XML-expression is null, the result is the null value.

Note:

1. Encoding in the serialized result: The serialized result is encoded with UTF-8. If XMLSERIALIZE is used with a character data type, and the INCLUDING XMLDECLARATION clause is specified, the resulting character string containing serialized XML might have an XML encoding declaration that does not match the code page of the character string. Following serialization, which uses UTF-8 encoding, the character string that is returned from the server to the client is converted to the code page of the client, and that code page might be different from UTF-8.

   Therefore, applications should avoid direct use of XMLSERIALIZE INCLUDING XMLDECLARATION that return character string types and should retrieve XML values directly into host variables to maintain the match between the external code page and the encoding in the XML declaration. If XMLSERIALIZE must be used in this situation, a BLOB type should be specified to avoid code page conversion.

2. Syntax alternative: XMLCLOB(XML-expression) can be specified in place of XMLSERIALIZE(XML-expression AS CLOB(2G)). It is supported only for compatibility with previous DB2 releases.
XMLTEXT

The schema is SYSIBM. The function name cannot be specified as a qualified name.

The XMLTEXT function returns an XML value with a single XQuery text node having the input argument as the content.

\( \text{string-expression} \)

An expression whose value has a character string type: CHAR, VARCHAR or CLOB.

The data type of the result is XML. If the result of \( \text{string-expression} \) can be null, the result can be null; if the input value is null, the result is the null value. If the result of \( \text{string-expression} \) is an empty string, the result value is an empty text node.

Examples:

- Create a simple XMLTEXT query.

  ```sql
  VALUES(
    XMLTEXT('The stock symbol for Johnson&Johnson is JNJ.'
  )
  )
  ```

  This query produces the following serialized result:

  1
  ---------------------------------------------------
  The stock symbol for Johnson&Johnson is JNJ.

  Note that the ‘&’ sign is mapped to ‘&amp;’ when a text node is serialized.

- Use XMLTEXT with XMLAGG to construct mixed content. Suppose that the content of table T is as follows:

<table>
<thead>
<tr>
<th>seqno</th>
<th>plaintext</th>
<th>emphtext</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>This query shows how to construct mixed content</td>
<td>XMLTEXT</td>
</tr>
<tr>
<td>2</td>
<td>using XMLAGG and XMLTEXT. Without XMLTEXT will not have text nodes to group with other nodes, mixed content therefore, cannot generate</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>SELECT XMLELEMENT(</td>
<td></td>
</tr>
<tr>
<td></td>
<td>NAME &quot;para&quot;, XMLAGG(</td>
<td></td>
</tr>
<tr>
<td></td>
<td>XMLCONCAT(</td>
<td></td>
</tr>
<tr>
<td></td>
<td>XMLTEXT(</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PLAINTEXT</td>
<td></td>
</tr>
</tbody>
</table>
  |       |         )
  |       |     ),
  |       |     XMLELEMENT( |
  |       |         NAME "emphasis", EMPHTEXT |
  |       |     )
  |       |   ) AS "result"
  |       | FROM T
  |

  This query produces the following result:

  <para>This query shows how to construct <emphasis>mixed content</emphasis>

  XMLTEXT

Chapter 3. Functions 657
using XMLAGG and XMLTEXT. Without <emphasis>XMLTEXT</emphasis>, XMLAGG
will not have text nodes to group with other nodes, therefore, cannot generate
<emphasis>mixed content</emphasis>.
</para>
XMLVALIDATE

XMLVALIDATE

XML-validate-according-to-clause:

ACCORDING TO XMLSCHEMA

ID—XML-schema-name

URI—XML-uri1

NO NAMESPACE

LOCATION—XML-uri2

XML-valid-element-clause

XML-valid-element-clause:

NAMESPACE—XML-uri3

NO NAMESPACE

ELEMENT—XML-element-name

The schema is SYSIBM. The function name cannot be specified as a qualified name.

The XMLVALIDATE function returns a copy of the input XML value augmented with information obtained from XML schema validation, including default values.

DOCUMENT

Specifies that the XML value resulting from XML-expression must be a well-formed XML document that conforms to XML Version 1.0 (SQLSTATE 2200M).

XML-expression

An expression that returns a value of data type XML. If XML-expression is an XML host variable or an implicitly or explicitly typed parameter marker, the function performs a validating parse that strips ignorable whitespace and the CURRENT IMPLICIT XMLPARSE OPTION setting is not considered.

XML-validate-according-to-clause

Specifies the information that is to be used when validating the input XML value.

ACCORDING TO XMLSCHEMA

Indicates that the XML schema information for validation is explicitly specified. If this clause is not included, the XML schema information must be provided in the content of the XML-expression value.

ID XML-schema-name

Specifies an SQL identifier for the XML schema that is to be used for validation. The name, including the implicit or explicit SQL schema qualifier, must uniquely identify an existing XML schema in the XML schema repository at the current server. If no XML schema by this name exists in the implicitly or explicitly specified SQL schema, an error is returned (SQLSTATE 42704).

URI XML-uri1

Specifies the target namespace URI of the XML schema that is to be used for validation. The value of XML-uri1 specifies a URI as a
XMLVALIDATE

character string constant that is not empty. The URI must be the target
namespace of a registered XML schema (SQLSTATE 4274A) and, if no
LOCATION clause is specified, it must uniquely identify the registered
XML schema (SQLSTATE 4274B).

NO NAMESPACE
Specifies that the XML schema for validation has no target namespace.
The target namespace URI is equivalent to an empty character string
that cannot be specified as an explicit target namespace URI.

LOCATION XML-uri2
Specifies the XML schema location URI of the XML schema that is to
be used for validation. The value of XML-uri2 specifies a URI as a
character string constant that is not empty. The XML schema location
URI, combined with the target namespace URI, must identify a
registered XML schema (SQLSTATE 4274A), and there must be only
one such XML schema registered (SQLSTATE 4274B).

XML-valid-element-clause
Specifies that the XML value in XML-expression must have the specified
element name as the root element of the XML document.

NAMESPACE XML-uri3 or NO NAMESPACE
Specifies the target namespace for the element that is to be validated. If
neither clause is specified, the specified element is assumed to be in the
same namespace as the target namespace of the registered XML schema
that is to be used for validation.

NAMESPACE XML-uri3
Specifies the namespace URI for the element that is to be validated. The
value of XML-uri3 specifies a URI as a character string constant
that is not empty. This can be used when the registered XML schema
that is to be used for validation has more than one namespace.

NO NAMESPACE
Specifies that the element for validation has no target namespace. The
target namespace URI is equivalent to an empty character string which
cannot be specified as an explicit target namespace URI.

ELEMENT xml-element-name
Specifies the name of a global element in the XML schema that is to be
used for validation. The specified element, with implicit or explicit
namespace, must match the root element of the value of
XML-expression (SQLSTATE 22535 or 22536).

The data type of the result is XML. If the value of XML-expression can be null, the
result can be null; if the value of XML-expression is null, the result is the null value.

The XML validation process is performed on a serialized XML value. Because
XMLVALIDATE is invoked with an argument of type XML, this value is
automatically serialized prior to validation processing with the follow two
exceptions.

• If the argument to XMLVALIDATE is an XML host variable or an implicitly or
explicitly typed parameter marker, then a validating parse operation is
performed on the input value (no implicit non-validating parse is performed and
CURRENT IMPLICIT XMLPARSE OPTION setting is not considered).

• If the argument to XMLVALIDATE is an XMLPARSE invocation using the option
PRESERVE WHITESPACE, then the XML parsing and XML validation of the
document may be combined into a single validating parse operation.
If an XML value has previously been validated, the annotated type information from the previous validation is removed by the serialization process. However, any default values and entity expansions from the previous validation remain unchanged. If validation is successful, all ignorable whitespace characters are stripped from the result.

To validate a document whose root element does not have a namespace, an xsi:noNamespaceSchemaLocation attribute must be present on the root element.

Note:

1. Determining the XML schema: The XML schema can be specified explicitly as part of XMLVALIDATE invocation, or determined from the XML schema information in the input XML value. If the XML schema information is not specified during invocation, the target namespace and the schema location in the input XML value are used to identify the registered schema for validation. If an explicit XML schema is not specified, the input XML value must contain an XML schema information hint (SQLSTATE 2200M). Explicit or implicit XML schema information must identify a registered XML schema (SQLSTATE 42704, 4274A, or 22532), and there must be only one such registered XML schema (SQLSTATE 4274B or 22533).

2. XML schema authorization: The XML schema used for validation must be registered in the XML schema repository prior to use. The privileges held by the authorization ID of the statement must include at least one of the following:
   - USAGE privilege on the XML schema that is to be used during validation
   - DBADM authority

Examples:

- Validate using the XML schema identified by the XML schema hint in the XML instance document.
  
  ```sql
  INSERT INTO T1(XMLCOL) 
  VALUES (XMLVALIDATE(?))
  ```

  Assume that the input parameter marker is bound to an XML value that contains the XML schema information.

  ```xml
  <po:order 
  xmlns:po='http://my.world.com' 
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" 
  xsi:schemaLocation="http://my.world.com/world.xsd" > 
  ... 
  </po:order>
  ```

  Further, assume that the XML schema that is associated with the target namespace “http://my.world.com” and by schemaLocation hint “http://my.world.com/world.xsd” is found in the XML schema repository.

  Based on these assumptions, the input XML value will be validated according to that XML schema.

- Validate using the XML schema identified by the SQL name PODOCS.WORLDPO.
  
  ```sql
  INSERT INTO T1(XMLCOL) 
  VALUES (XMLVALIDATE(?
  
  ? ACCORDING TO XMLSCHEMA ID PODOCS.WORLDPO 
  )
  )
  ```
XMLVALIDATE

Assuming that the XML schema that is associated with SQL name
FOO.WORLDPO is found in the XML repository, the input XML value will be
validated according to that XML schema.

- Validate a specified element of the XML value.

```sql
INSERT INTO T1(XMLCOL)
VALUES (
    XMLVALIDATE(
        ? ACCORDING TO XMLSCHEMA ID FOO.WORLDPO
        NAMESPACE 'http://my.world.com/Mary'
        ELEMENT "po"
    )
)
```

Assuming that the XML schema that is associated with SQL name
FOO.WORLDPO is found in the XML repository, the XML schema will be
validated against the element "po", whose namespace is 'http://my.world.com/
Mary'.

- XML schema is identified by target namespace and schema location.

```sql
INSERT INTO T1(XMLCOL)
VALUES (
    XMLVALIDATE(
        ? ACCORDING TO XMLSCHEMA URI 'http://my.world.com'
        LOCATION 'http://my.world.com/world.xsd'
    )
)
```

Assuming that an XML schema associated with the target namespace
"http://my.world.com" and by schemaLocation hint "http://my.world.com/
world.xsd" is found in the XML schema repository, the input XML value will be
validated according to that schema.
The schema is SYSIBM.

The XMLXSROBJECTID function returns the XSR object identifier of the XML schema used to validate the XML document specified in the argument. The XSR object identifier is returned as a BIGINT value and provides the key to a single row in SYSCAT.XSROBJECTS.

\textit{xml-value-expression}

Specifies an expression that results in a value with a data type of XML. The resulting XML value must be an XML sequence with a single item that is an XML document or the null value (SQLSTATE 42815). If the argument is null, the function returns null. If \textit{xml-value-expression} does not specify a validated XML document, the function returns 0.

\textbf{Note:}

1. The XML schema corresponding to an XSR object ID returned by the function might no longer exist, because an XML schema can be dropped without affecting XML values that were validated using the XML schema. Therefore, queries that use the XSR object ID to fetch further XML schema information from the catalog views might return an empty result set.

2. Applications can use the XSR object identifier to retrieve additional information about the XML schema. For example, the XSR object identifier can be used to return the individual XML schema documents that make up a registered XML schema from SYSCAT.SYXSROBJECTCOMPONENTS, and the hierarchy of XML schema documents in the XML schema from SYSCAT.XSROBJECTHIERARCHIES.

\textbf{Examples:}

- Retrieve the XML schema identifier for the XML document XMLDOC stored in the table MYTABLE.
  \begin{verbatim}
  SELECT XMLXSROBJECTID(XMLDOC) FROM MYTABLE
  \end{verbatim}

- Retrieve the XML schema documents associated with the XML document that has a specific ID (in this case where DOCKEY = 1) in the table MYTABLE, including the hierarchy of the XML schema documents that make up the XML schema.
  \begin{verbatim}
  SELECT H.HTYPE, C.TARGETNAMESPACE, C.COMPONENT
  FROM SYSCAT.XSROBJECTCOMPONENTS C, SYSCAT.XSROBJECTHIERARCHIES H
  WHERE C.OBJECTID =
    (SELECT XMLXSROBJECTID(XMLDOC) FROM MYTABLE
     WHERE DOCKEY = 1)
  AND C.OBJECTID = H.OBJECTID
  \end{verbatim}
XSLTRANSFORM

Use XSLTRANSFORM to convert XML data into other formats, including the conversion of XML documents that conform to one XML schema into documents that conform to another schema.

**xml-document**

An expression that returns a well-formed XML document with a data type of XML, CHAR, VARCHAR, CLOB, or BLOB. This is the document that is transformed using the XSL style sheet specified in **xsl-stylesheet**.

**Note:**

The XML document must at minimum be single-rooted and well-formed.

**xsl-stylesheet**

An expression that returns a well-formed XML document with a data type of XML, CHAR, VARCHAR, CLOB, or BLOB. The document is an XSL style sheet that conforms to the W3C XSLT Version 1.0 Recommendation. Style sheets

**data-type:**

The schema is SYSIBM. This function cannot be specified as a qualified name.

The XSLTRANSFORM function transforms an XML document into a different data format. The data can be transformed into any form possible for the XSLT processor, including but not limited to XML, HTML, or plain text.

All paths used by XSLTRANSFORM are internal to the database system. This command cannot currently be used directly with files or stylesheets residing in an external file system.
incorporating XQUERY statements or the xsl:include declaration are not supported. This stylesheet is applied to transform the value specified in xml-document.

xsl-parameters
An expression that returns a well-formed XML document or null with a data type of XML, CHAR, VARCHAR, CLOB, or BLOB. This is a document that provides parameter values to the XSL stylesheet specified in xsl-stylesheet. The value of the parameter can be specified as an attribute, or as a text node.

The syntax of the parameter document is as follows:
<params xmlns="http://www.ibm.com/XSLTransformParameters">
  <param name="..." value="..."/>
  <param name="...">enter value here</param>
...
</params>

Note:
The stylesheet document must have xsl:param element(s) in it with name attribute values that match the ones specified in the parameter document.

AS data-type
Specifies the result data type. The implicit or explicit length attribute of the specified result data type must be sufficient to contain the transformed output (SQLSTATE 22001). The default result data type is CLOB(2G).

Note:
If either the xml-document argument or the xsl-stylesheet argument is null, the result will be null.

Code page conversion might occur when storing any of the above documents in a CHAR, VARCHAR, or CLOB column, which might result in a character loss.

Example

This example illustrates how to use XSLT as a formatting engine. To get set up, first insert the two example documents below into the database.

```xml
INSERT INTO XML_TAB VALUES
  (1,
    '<?xml version="1.0"?>
    <students xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:noNamespaceSchemaLocation="/home/steffen/xsd/xslt.xsd">
      <student studentID="1" firstName="Steffen" lastName="Siegmund"
        age="23" university="Rostock"/>
    </students>',
    '<?xml version="1.0" encoding="UTF-8"?>
    <xsl:stylesheet version="1.0"
      xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
      <xsl:param name="headline"/>
      <xsl:param name="showUniversity"/>
      <xsl:template match="students">
        <html>
        <head/>
        <body/>
        <h1><xsl:apply-templates select="/students/student/studentID"/>
        <table border="1">
          <th>
            <tr>
```
In this example, the output is HTML and the parameters influence only what HTML is produced and what data is brought over to it. As such it illustrates the use of XSLT as a formatting engine for end-user output.
Usage note:

This function is **not** intended for high-performance applications and **cannot** replace similar functionality in the application server.
The YEAR function returns the year part of a value.

The argument must be a DATE, TIMESTAMP, date duration, timestamp duration, or a valid character string representation of a date or timestamp that is not a CLOB. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

The result of the function is a large integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The other rules depend on the data type of the argument specified:

- If the argument is a DATE, TIMESTAMP, or valid string representation of a date or timestamp:
  - The result is the year part of the value, which is an integer between 1 and 9999.
- If the argument is a date duration or timestamp duration:
  - The result is the year part of the value, which is an integer between -9999 and 9999. A nonzero result has the same sign as the argument.

**Examples**

- Select all the projects in the PROJECT table that are scheduled to start (PRSTDATE) and end (PRENDATE) in the same calendar year.
  ```sql
  SELECT * FROM PROJECT
  WHERE YEAR(PRSTDATE) = YEAR(PRENDATE)
  ```

- Select all the projects in the PROJECT table that are scheduled to take less than one year to complete.
  ```sql
  SELECT * FROM PROJECT
  WHERE YEAR(PRENDATE - PRSTDATE) < 1
  ```

**Table functions**

A table function can be used only in the FROM clause of a statement. Table functions return columns of a table, resembling a table created through a simple CREATE TABLE statement. Table functions can be qualified with a schema name.
The schema is SYSPROC.

The BASE_TABLE function returns both the object name and schema name of the object found after any alias chains have been resolved. The specified objectname (and objectschema) are used as the starting point of the resolution. If the starting point does not refer to an alias, the schema name and the unqualified name of the starting point are returned. The function returns a single row table consisting of the following columns:

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BASESCHEMA</td>
<td>VARCHAR(128)</td>
<td>Schema name of the object found after any alias chains have been resolved. Matches objectschema if no matching alias was found.</td>
</tr>
<tr>
<td>BASENAME</td>
<td>VARCHAR(128)</td>
<td>Unqualified name of the object found after any alias chains have been resolved. Matches objectname if no matching alias was found. The name may identify a table, a view, or an undefined object.</td>
</tr>
</tbody>
</table>

**objectschema**

A character expression representing the schema used to qualify the supplied objectname value before resolution. objectschema must have a data type of CHAR or VARCHAR and a length greater than 0 and less than 129 bytes.

**objectname**

A character expression representing the unqualified name to be resolved. objectname must have a data type of CHAR or VARCHAR and a length greater than 0 and less than 129 bytes.

**Note:** The BASE_TABLE table function improves performance in partitioned database configurations by avoiding the unnecessary communication that occurs between the coordinator partition and catalog partition when using the TABLE_SCHEMA and TABLE_NAME scalar functions.

**Example**

The following statement using the TABLE_SCHEMA and TABLE_NAME functions is written as:

```
SELECT COLCOUNT INTO :H00030
FROM SYSCAT.TABLES
WHERE OWNER = TABLE_SCHEMA(:H00031 ,:H00032 )
AND TABNAME = TABLE_NAME(:H00031 ,:H00032 )
```

The equivalent statement using the BASE_TABLE function can be written as:
```sql
SELECT COLCOUNT INTO :H00030
FROM SYSCAT.TABLES A, TABLE(SYSPROC.BASE_TABLE(:H00032, :H00031)) AS B
WHERE A.OWNER = B.BASESCHEMA
AND A.TABNAME = B.BASENAME
```
The schema is SYSIBM.

The UNNEST function returns a result table that includes a row for each element of the specified array. If there are multiple ordinary array arguments specified, the number of rows will match the array with the largest cardinality.

**ordinary-array-variable**

An SQL variable, SQL parameter, or global variable of an ordinary array type, or a CAST specification of a parameter marker to an ordinary array type.

**associative-array-variable**

An SQL variable, SQL parameter, or global variable of an associative array type, or a CAST specification of a parameter marker to an associative array type.

Names for the result columns produced by the UNNEST function can be provided as part of the required correlation-clause of the collection-derived-table clause.

The UNNEST function can only be used in a collection-derived-table clause in a context where arrays are supported (SQLSTATE 42887).

The result table depends on the input arguments.

- If a single ordinary array argument is specified:
  - If the array element is not a row data type, the result is a single column table with a column data type that matches the array element data type.
  - If the array element is a row data type, the result is a table with one column for each row field in the element data type. The result table column data types match the corresponding array element row field data types.
- If more than one ordinary array argument is specified and none of the array elements have a row data type, the first array provides the first column in the result table, the second array provides the second column, and so on. The data type of each column matches the data type of the array elements of the corresponding array argument. If the cardinalities of the arrays are not identical, the cardinality of the resulting table is the same as the array with the largest cardinality. The column values in the table are set to the null value for all rows whose array index value is greater than the cardinality of the corresponding array. In other words, if each array is viewed as a table with two columns, one for the array indexes and one for the data, then UNNEST performs an OUTER JOIN among the arrays, using equality on the array indexes as a join predicate.
- If a single associative array argument is specified:
  - If the array element is not a row data type, the result is a table with 2 columns where the first column data type matches the array index data type and the second column data type matches the array element data type.
  - If the array element is a row data type, the result is a table with one more column than the number of fields in the row data type, where the first column data type matches the array index data type and the remaining column data types match the array element row field data types.
UNNEST

- An error is returned (SQLSTATE 42884):
  - If more than one associative array argument is specified.
  - If more than one array argument is specified and at least one of the arrays 
    has a element data type that is a row type.
  - If both ordinary array arguments and associative array arguments are 
    specified.

Names for the result columns produced by the UNNEST function can be provided 
as part of the required correlation-clause of collection-derived-table.

This special table function is only used in collection-derived-table of table-reference in 
a FROM clause.

If more than one array is provided and at least one of the arguments is an 
associative array, an error is returned (SQLSTATE 42884).

If the WITH ORDINALITY clause is used when unnesting an associative array, an 
error is returned (SQLSTATE 428HT).

Examples

- Assume the ordinary array variable RECENT_CALLS of array type PHONENUMBERS 
  contains only the three element values 9055553907, 4165554213, and 
  4085553678. The following query:

  ```sql
  SELECT T.ID, T.NUM
  FROM UNNEST(RECENT_CALLS) WITH ORDINALITY AS T(NUM, ID)
  ```

  returns a table formatted as follows:

<table>
<thead>
<tr>
<th>ID</th>
<th>NUM</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>9055553907</td>
</tr>
<tr>
<td>2</td>
<td>4165554213</td>
</tr>
<tr>
<td>3</td>
<td>4085553678</td>
</tr>
</tbody>
</table>

- Return the list of personal phone numbers from the array variable PHONELIST of 
  array type PERSONAL_PHONENUMBERS along with the index string values The 
  following query:

  ```sql
  SELECT T.ID, T.PHONE
  FROM UNNEST(PHONELIST) AS T(ID, PHONE)
  ```

  returns a table formatted as follows:

<table>
<thead>
<tr>
<th>ID</th>
<th>PHONE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Home</td>
<td>4163053745</td>
</tr>
<tr>
<td>Work</td>
<td>4163053746</td>
</tr>
<tr>
<td>Mom</td>
<td>4164789683</td>
</tr>
</tbody>
</table>
XMLTABLE

xmlnamespaces-declaration

(row-xquery-expression-constant)

BY REF

row-xquery-argument

COLUMNS

xml-table-regular-column-definition

xml-table-ordinality-column-definition

row-xquery-argument:

xquery-variable-expression AS identifier

BY REF

xml-table-regular-column-definition:

column-name data-type

BY REF

default-clause

PATH column-xquery-expression-constant

xml-table-ordinality-column-definition:

column-name FOR ORDINALITY

Notes:
1. The xml-table-ordinality-column-definition clause must not be specified more than once (SQLSTATE 42614).
2. The data type of the expression cannot be DECFLOAT.

The schema is SYSIBM. The function name cannot be specified as a qualified name.

The XMLTABLE function returns a result table from the evaluation of XQuery expressions, possibly using specified input arguments as XQuery variables. Each sequence item in the result sequence of the row XQuery expression represents a row of the result table.

xmlnamespaces-declaration

Specifies one or more XML namespace declarations that become part of the static context of the row-xquery-expression-constant and the column-xquery-expression-constant. The set of statically known namespaces for XQuery expressions which are arguments of XMLTABLE is the combination of the pre-established set of statically known namespaces and the namespace
declarations specified in this clause. The XQuery prolog within an XQuery
expression may override these namespaces.

If `xmlnamespaces-declaration` is not specified, only the pre-established set of
statically known namespaces apply to the the XQuery expressions.

**row-xquery-expression-constant**
Specifies an SQL character string constant that is interpreted as an XQuery
expression using supported XQuery language syntax. The constant string is
converted directly to UTF-8 without conversion to the database or section code
page. The XQuery expression executes using an optional set of input XML
values, and returns an output XQuery sequence where a row is generated for
each item in the sequence. The value for `row-xquery-expression-constant` must not
be an empty string or a string of all blanks (SQLSTATE 10505).

**PASSING**
Specifies input values and the manner in which these values are passed to the
XQuery expression specified by `row-xquery-expression-constant`. By default, every
unique column name that is in the scope where the function is invoked is
implicitly passed to the XQuery expression using the name of the column as
the variable name. If an `identifier` in a specified row-xquery-argument matches
an in-scope column name, then the explicit row-xquery-argument is passed to
the XQuery expression overriding that implicit column.

**BY REF**
Specifies that any XML input arguments are, by default, passed by
reference. When XML values are passed by reference, the XQuery
evaluation uses the input node trees, if any, directly from the specified
input expressions, preserving all properties, including the original node
identities and document order. If two arguments pass the same XML value,
node identity comparisons and document ordering comparisons involving
some nodes contained between the two input arguments might refer to
nodes within the same XML node tree.

This clause has no impact on how non-XML values are passed. The
non-XML values create a new copy of the value during the cast to XML.

**row-xquery-argument**
Specifies an argument that is to be passed to the XQuery expression
specified by `row-xquery-expression-constant`. An argument specifies a value
and the manner in which that value is to be passed. The argument
includes an SQL expression that is evaluated before passing the result to
the XQuery expression.

- If the resulting value is of type XML, it becomes an `input-xml-value`. A
  null XML value is converted to an XML empty sequence.
- If the resulting value is not of type XML, it must be castable to the XML
data type. A null value is converted to an XML empty sequence. The
  converted value becomes an `input-xml-value`.

When the `row-xquery-expression-constant` is evaluated, an XQuery variable is
presented with a value equal to `input-xml-value` and a name specified by
the AS clause.

**xquery-variable-expression**
Specifies an SQL expression whose value is available to the XQuery
expression specified by `row-xquery-expression-constant` during execution.
The expression cannot contain a NEXT VALUE expression, PREVIOUS
XMLTABLE

VALUE expression (SQLSTATE 428F9), or an OLAP function (SQLSTATE 42903). The data type of the expression cannot be DEC_FLOAT.

**AS** *identifier*

Specifies that the value generated by *xquery-variable-expression* will be passed to *row-xquery-expression-constant* as an XQuery variable. The variable name will be *identifier*. The leading dollar sign ($) that precedes variable names in the XQuery language is not included in *identifier*. The identifier must be a valid XQuery variable name and is restricted to an XML NCName. The identifier must not be greater than 128 bytes in length. Two arguments within the same PASSING clause cannot use the same identifier (SQLSTATE 42711).

**BY REF**

Indicates that an XML input value is to be passed by reference. When XML values are passed by reference, the XQuery evaluation uses the input node trees, if any, directly from the specified input expressions, preserving all properties, including the original node identities and document order. If two arguments pass the same XML value, node identity comparisons and document ordering comparisons involving some nodes contained between the two input arguments might refer to nodes within the same XML node tree. If BY REF is not specified following an *xquery-expression-variable*, XML arguments are passed by way of the default passing mechanism that is provided through the syntax that follows the PASSING keyword. This option cannot be specified for non-XML values (SQLSTATE 42636). When a non-XML value is passed, the value is converted to XML; this process creates a copy.

**COLUMNS**

Specifies the output columns of the result table. If this clause is not specified, a single unnamed column of data type XML is returned by reference, with the value based on the sequence item from evaluating the XQuery expression in the *row-xquery-expression-constant* (equivalent to specifying PATH "."). To reference the result column, a *column-name* must be specified in the *correlation-clause* following the function.

**xml-table-regular-column-definition**

Specifies the output columns of the result table including the column name, data type, XML passing mechanism and an XQuery expression to extract the value from the sequence item for the row.

*column-name*

Specifies the name of the column in the result table. The name cannot be qualified and the same name cannot be used for more than one column of the table (SQLSTATE 42711).

*data-type*

Specifies the data type of the column. See CREATE TABLE for the syntax and a description of types available. A *data-type* may be used in XMLTable if there is a supported XMLCAST from the XML data type to the specified *data-type*.

**BY REF**

Specifies that XML values are returned by reference for columns of data type XML. By default, XML values are returned BY REF. When XML values are returned by reference, the XML value includes the input node trees, if any, directly from the result values, and preserves
all properties, including the original node identities and document order. This option cannot be specified for non-XML columns (SQLSTATE 42636). When a non-XML column is processed, the value is converted from XML; this process creates a copy.

default-clause
Specifies a default value for the column. See CREATE TABLE for the syntax and a description of the default-clause. For XMLTABLE result column, the default is applied when the processing the XQuery expression contained in column-xquery-expression-constant returns an empty sequence.

PATH column-xquery-expression-constant
Specifies an SQL character string constant that is interpreted as an XQuery expression using supported XQuery language syntax. The constant string is converted directly to UTF-8 without conversion to the database or section code page. The column-xquery-expression-constant specifies an XQuery expression that determines the column value with respect to an item that is the result of evaluating the XQuery expression in row-xquery-expression-constant. Given an item from the result of processing the row-xquery-expression-constant as the externally provided context item, the column-xquery-expression-constant is evaluated, returning an output sequence. The column value is determined based on this output sequence as follows.

• If the output sequence contains zero items, the default-clause provides the value of the column.
• If an empty sequence is returned and no default-clause was specified, a null value is assigned to the column.
• If a non-empty sequence is returned, the value is XMLCAST to the data-type specified for the column. An error could be returned from processing this XMLCAST.

The value for column-xquery-expression-constant must not be an empty string or a string of all blanks (SQLSTATE 10505). If this clause is not specified, the default XQuery expression is simply the column-name.

xml-table-ordinality-column-definition
Specifies the ordinality column of the result table.

column-name
Specifies the name of the column in the result table. The name cannot be qualified and the same name cannot be used for more than one column of the table (SQLSTATE 42711).

FOR ORDINALITY
Specifies that column-name is the ordinality column of the result table. The data type of this column is BIGINT. The value of this column in the result table is the sequential number of the item for the row in the resulting sequence from evaluating the XQuery expression in row-xquery-expression-constant.

If the evaluation of any of the XQuery expressions results in an error, then the XMLTABLE function returns the XQuery error (SQLSTATE class ‘10’).

Examples:
• List as a table result the purchase order items for orders with a status of ‘NEW’.
User-defined functions

User-defined functions (UDFs) are extensions or additions to the existing built-in functions of the SQL language. A user-defined function can be a scalar function, which returns a single value each time it is called; an aggregate function, which is passed a set of like values and returns a single value for the set; a row function, which returns one row; or a table function, which returns a table.

A number of user-defined functions are provided in the SYSFUN and SYSPROC schemas.

A UDF can be an aggregate function only if it is sourced on an existing aggregate function. A UDF is referenced by means of a qualified or unqualified function name, followed by parentheses enclosing the function arguments (if any). A user-defined column or scalar function registered with the database can be referenced in the same contexts in which any built-in function can appear. A user-defined row function can be referenced only implicitly when registered as a transform function for a user-defined type. A user-defined table function registered with the database can be referenced only in the FROM clause of a SELECT statement.

Function arguments must correspond in number and position to the parameters specified for the user-defined function when it was registered with the database. In addition, the arguments must be of data types that are promotable to the data types of the corresponding defined parameters.

The result of the function is specified in the RETURNS clause. The RETURNS clause, defined when the UDF was registered, determines whether or not a function is a table function. If the RETURNS NULL ON NULL INPUT clause is specified (or defaulted to) when the function is registered, the result is null if any argument is null. In the case of table functions, this is interpreted to mean a return table with no rows (that is, an empty table).

See “Row expressions” for more information on rules and row data types.

Following are some examples of user-defined functions:
User-defined functions

- A scalar UDF called ADDRESS extracts the home address from resumes stored in script format. The ADDRESS function expects a CLOB argument and returns a VARCHAR(4000) value:

  ```sql
  SELECT EMPNO, ADDRESS(RESUME) FROM EMP_RESUME
  WHERE RESUME_FORMAT = 'SCRIPT'
  ```

- Table T2 has a numeric column A. Invoking the scalar UDF called ADDRESS from the previous example:

  ```sql
  SELECT ADDRESS(A) FROM T2
  ```
  raises an error (SQLSTATE 42884), because no function with a matching name and with a parameter that is promotable from the argument exists.

- A table UDF called WHO returns information about the sessions on the server machine that were active at the time that the statement is executed. The WHO function is invoked from within a FROM clause that includes the keyword TABLE and a mandatory correlation variable. The column names of the WHO() table were defined in the CREATE FUNCTION statement.

  ```sql
  SELECT ID, START_DATE, ORIG_MACHINE
  FROM TABLE( WHO() ) AS QQ
  WHERE START_DATE LIKE 'MAY%'
  ```
Chapter 4. Procedures

Procedures overview

A procedure is an application program that can be started through the SQL CALL statement. The procedure is specified by a procedure name, which may be followed by arguments that are enclosed within parentheses.

The argument or arguments of a procedure are individual scalar values, which can be of different types and can have different meanings. The arguments can be used to pass values into the procedure, receive return values from the procedure, or both.

User-defined procedures are procedures that are registered to a database in SYSCAT.ROUTINES, using the CREATE PROCEDURE statement. One such set of functions is provided with the database manager, in a schema called SYSFUN, and another in a schema called SYSPROC.

Procedures can be qualified with the schema name.

XSR_ADDSCHEMADOC

The schema is SYSPROC.

Each XML schema in the XML schema repository (XSR) can consist of one or more XML schema documents. Where an XML schema consists of multiple documents, the XSR_ADDSCHEMADOC stored procedure is used to add every XML schema other than the primary XML schema document.

Authorization

The authorization ID of the caller of the procedure must be the owner of the XSR object as recorded in the catalog view SYSCAT.XSROBJECTS.

rschema

An input argument of type VARCHAR (128) that specifies the SQL schema for the XML schema. The SQL schema is one part of the SQL identifier used to identify this XML schema in the XSR, which is to be moved to the complete state. (The other part of the SQL identifier is supplied by the name argument.) This argument can have a NULL value, which indicates that the default SQL schema, as defined in the CURRENT SCHEMA special register, is used. Rules for valid characters and delimiters that apply to any SQL identifier also apply to this argument. XSR objects will not experience name collisions with database objects that exist outside of the XSR, because XSR objects occur in a different namespace than objects outside of the XML schema repository.

name

An input argument of type VARCHAR (128) that specifies the name of the
XML schema. The complete SQL identifier for the XML schema is `rschema.name`. The XML schema name must already exist as a result of calling the `XSR_REGISTER` stored procedure, and XML schema registration cannot yet be completed. This argument cannot have a NULL value. Rules for valid characters and delimiters that apply to any SQL identifier also apply to this argument.

`schemalocation`

An input argument of type VARCHAR (1000), which can have a NULL value, that indicates the schema location of the primary XML schema document to which the XML schema document is being added. This argument is the external name of the XML schema, that is, the primary document can be identified in the XML instance documents with the `xsi:schemaLocation` attribute.

`content`

An input parameter of type BLOB (30M) that contains the content of the XML schema document being added. This argument cannot have a NULL value; an XML schema document must be supplied.

`docproperty`

An input parameter of type BLOB (5M) that indicates the properties for the XML schema document being added. This parameter can have a NULL value; otherwise, the value is an XML document.

Example:

```sql
CALL SYSPROC.XSR_ADDSCHEMADOC('user1', 'POschema', 'http://myPOschema/address.xsd', :content_host_var, 0)
```

The schema is SYSPROC.

The `XSR_COMPLETE` procedure is the final stored procedure to be called as part of the XML schema registration process, which registers XML schemas with the XML schema repository (XSR). An XML schema is not available for validation until the schema registration completes through a call to this stored procedure.

**Authorization:**

The authorization ID of the caller of the procedure must be the owner of the XSR object as recorded in the catalog view `SYSCAT.XSROBJECTS`.

`rschema`

An input argument of type VARCHAR (128) that specifies the SQL schema for the XML schema. The SQL schema is one part of the SQL identifier used to identify this XML schema in the XSR, which is to be moved to the complete state. (The other part of the SQL identifier is supplied by the `name` argument.)

This argument can have a NULL value, which indicates that the default SQL
schema, as defined in the CURRENT SCHEMA special register, is used. Rules for valid characters and delimiters that apply to any SQL identifier also apply to this argument. XSR objects will not experience name collisions with database objects that exist outside of the XSR, because XSR objects occur in a different namespace than objects outside of the XML schema repository.

**name**
An input argument of type VARCHAR (128) that specifies the name of the XML schema. The complete SQL identifier for the XML schema, for which a completion check is to be performed, is *rschema.name*. The XML schema name must already exist as a result of calling the XSR_REGISTER stored procedure, and XML schema registration cannot yet be completed. This argument cannot have a NULL value. Rules for valid characters and delimiters that apply to any SQL identifier also apply to this argument.

**schemaproperties**
An input argument of type BLOB (5M) that specifies properties, if any, associated with the XML schema. The value for this argument is either NULL, if there are no associated properties, or an XML document representing the properties for the XML schema.

**isusedfordecomposition**
An input parameter of type integer that indicates if an XML schema is to be used for decomposition. If an XML schema is to be used for decomposition, this value should be set to 1; otherwise, it should be set to zero.

Example:
```sql
CALL SYSPROC.XSR_COMPLETE('user1', 'POschema', :schemaproperty_host_var, 0)
```

The schema is SYSPROC.

The XSR_DTD procedure registers a document type declaration (DTD) with the XML schema repository (XSR).

**Authorization**
The authorization ID of the caller of the procedure must have at least one of the following:
- DBADM authority.
- IMPLICIT_SCHEMA database authority if the SQL schema does not exist.
- CREATEIN privilege if the SQL schema exists.

**rschema**
An input and output argument of type VARCHAR (128) that specifies the SQL schema for the DTD. The SQL schema is one part of the SQL identifier used to identify this DTD in the XSR. (The other part of the SQL identifier is supplied by the *name* argument.) This argument can have a NULL value, which indicates that the default SQL schema, as defined in the CURRENT SCHEMA
special register, is used. Rules for valid characters and delimiters that apply to any SQL identifier also apply to this argument. Relational schemas that begin with the string 'SYS' must not be used for this value. XSR objects will not experience name collisions with database objects that exist outside of the XSR, because XSR objects occur in a different namespace than objects outside of the XML schema repository.

**name**
An input and output argument of type VARCHAR (128) that specifies the name of the DTD. The complete SQL identifier for the DTD is `rschema.name` and should be unique among all objects in the XSR. This argument accepts a NULL value. When a NULL value is provided for this argument, a unique value is generated and stored within the XSR. Rules for valid characters and delimiters that apply to any SQL identifier also apply to this argument.

**systemid**
An input parameter of type VARCHAR (1000) that specifies the system identifier of the DTD. The system ID of the DTD should match the uniform resource identifier of the DTD in the DOCTYPE declaration of the XML instance document or in an ENTITY declaration (as prefixed by the SYSTEM keyword, if used). This argument cannot have a NULL value. The system ID can be specified together with a public ID.

**publicid**
An input parameter of type VARCHAR (1000) that specifies the public identifier of the DTD. The public ID of a DTD should match the uniform resource identifier of the DTD in the DOCTYPE declaration of the XML instance document or in an ENTITY declaration (as prefixed by the PUBLIC keyword, if used). This argument accepts a NULL value and should be used only if also specified in the DOCTYPE declaration of the XML instance document or in an ENTITY declaration.

**content**
An input parameter of type BLOB (30M) that contains the content of the DTD document. This argument cannot have a NULL value.

Example: Register the DTD identified by the system ID `http://www.test.com/person.dtd` and public ID `http://www.test.com/person`:

```sql
```

The schema is SYSPROC.

The `XSR_EXTENTITY` procedure registers an external entity with the XML schema repository (XSR).
Authorization

The authorization ID of the caller of the procedure must have at least one of the following:

- DBADM authority.
- IMPLICIT_SCHEMA database authority if the SQL schema does not exist.
- CREATEIN privilege if the SQL schema exists.

rschema
An input and output argument of type VARCHAR (128) that specifies the SQL schema for the external entity. The SQL schema is one part of the SQL identifier used to identify this external entity in the XSR. (The other part of the SQL identifier is supplied by the name argument.) This argument can have a NULL value, which indicates that the default SQL schema, as defined in the CURRENT SCHEMA special register, is used. Rules for valid characters and delimiters that apply to any SQL identifier also apply to this argument. Relational schemas that begin with the string 'SYS' must not be used for this value. XSR objects will not experience name collisions with database objects that exist outside of the XSR, because XSR objects occur in a different namespace than objects outside of the XML schema repository.

name
An input and output argument of type VARCHAR (128) that specifies the name of the external entity. The complete SQL identifier for the external entity is rschema.name and should be unique among all objects in the XSR. This argument accepts a NULL value. When a NULL value is provided for this argument, a unique value is generated and stored within the XSR. Rules for valid characters and delimiters that apply to any SQL identifier also apply to this argument.

systemid
An input parameter of type VARCHAR (1000) that specifies the system identifier of the external entity. The system ID of the external entity should match the uniform resource identifier of the external entity in the ENTITY declaration (as prefixed by the SYSTEM keyword, if used). This argument cannot have a NULL value. The system ID can be specified together with a public ID.

publicid
An input parameter of type VARCHAR (1000) that specifies the public identifier of the external entity. The public ID of an external entity should match the uniform resource identifier of the external entity in the ENTITY declaration (as prefixed by the PUBLIC keyword, if used). This argument accepts a NULL value and should be used only if also specified in the DOCTYPE declaration of the XML instance document or in an ENTITY declaration.

content
An input parameter of type BLOB (30M) that contains the content of the external entity document. This argument cannot have a NULL value.

Example: Register the external entities identified by the system identifiers http://www.test.com/food/chocolate.txt and http://www.test.com/food/cookie.txt:

CALL SYSPROC.XSR_EXTENTITY ( 'FOOD' ,
  'CHOCOLATE' ,
  'http://www.test.com/food/chocolate.txt' ,
  NULL ,
  :content_of_chocolate.txt_as_a_host_variable
)
CALL SYSPROC.XSR_EXTENTITY ('FOOD', 'COOKIE', 'http://www.test.com/food/cookie.txt', NULL, :content_of_cookie.txt_as_a_host_variable)

The schema is SYSPROC.

The XSR_REGISTER procedure is the first stored procedure to be called as part of the XML schema registration process, which registers XML schemas with the XML schema repository (XSR).

Authorization

The authorization ID of the caller of the procedure must have at least one of the following:
- DBADM authority.
- IMPLICIT_SCHEMA database authority if the SQL schema does not exist.
- CREATEIN privilege if the SQL schema exists.

rschema
An input and output argument of type VARCHAR (128) that specifies the SQL schema for the XML schema. The SQL schema is one part of the SQL identifier used to identify this XML schema in the XSR. (The other part of the SQL identifier is supplied by the name argument.) This argument can have a NULL value, which indicates that the default SQL schema, as defined in the CURRENT SCHEMA special register, is used. Rules for valid characters and delimiters that apply to any SQL identifier also apply to this argument. Relational schemas that begin with the string 'SYS' must not be used for this value. XSR objects will not experience name collisions with database objects that exist outside of the XSR, because XSR objects occur in a different namespace than objects outside of the XML schema repository.

name
An input and output argument of type VARCHAR (128) that specifies the name of the XML schema. The complete SQL identifier for the XML schema is rschema.name and should be unique among all objects in the XSR. This argument accepts a NULL value. When a NULL value is provided for this argument, a unique value is generated and stored within the XSR. Rules for valid characters and delimiters that apply to any SQL identifier also apply to this argument.

schemalocation
An input argument of type VARCHAR (1000), which can have a NULL value, that indicates the schema location of the primary XML schema document. This argument is the external name of the XML schema, that is, the primary document can be identified in the XML instance documents with the xsi:schemaLocation attribute.
XSR_REGISTER

**content**
An input parameter of type BLOB (30M) that contains the content of the primary XML schema document. This argument cannot have a NULL value; an XML schema document must be supplied.

**docproperty**
An input parameter of type BLOB (5M) that indicates the properties for the primary XML schema document. This parameter can have a NULL value; otherwise, the value is an XML document.

Example: The following example shows how to call the XSR_REGISTER procedure from the command line:
```
CALL SYSPROC.XSR_REGISTER(
   'user1',
   'POschema',
   'http://myPOschema/PO.xsd',
   :content_host_var,
   :docproperty_host_var)
```

Example: The following example shows how to call the XSR_REGISTER procedure from a Java application program:
```
stmt = con.prepareCall("CALL SYSPROC.XSR_REGISTER (?, ?, ?, ?, ?)"");
String xsrObjectName = "myschema1";
String xmlSchemaLocation = "po.xsd";
stmt.setNull(1, java.sql.Types.VARCHAR);
stmt.setString(2, xsrObjectName);
stmt.setString(3, xmlSchemaLocation);
stmt.setBinaryStream(4, buffer, (int)length);
stmt.setNull(5, java.sql.Types.BLOB);
stmt.registerOutParameter(1, java.sql.Types.VARCHAR);
stmt.registerOutParameter(2, java.sql.Types.VARCHAR);
stmt.execute();
```

XSR_UPDATE

The schema is SYSPROC.

The XSR_UPDATE stored procedure is used to evolve an existing XML schema in the XML schema repository (XSR). This enables you to modify or extend an existing XML schema so that it can be used to validate both already existing and newly inserted XML documents.

The original XML schema and the new XML schema specified as arguments to XSR_UPDATE must both be registered and completed in the XSR before the procedure is called. These XML schemas must also be compatible. For details about the compatibility requirements see Compatibility requirements for evolving an XML schema.

**Authorization**

The privileges held by the authorization ID of the caller of the procedure must include at least one of the following:
- DBADM authority.
• SELECT privilege on the catalog views SYSCAT.XSROBJECTS and
  SYSCAT.XSROBJECTCOMPONENTS and one of the following sets of privileges:
  – OWNER of the XML schema specified by the SQL schema rschema1 and the
    object name name1
  – ALTERIN privilege on the SQL schema specified by the rschema1 argument
    and, if the dropnewschema argument is not equal to zero, DROPIN privilege on
    the SQL schema specified by the rschema2 argument.

rschema1
An input argument of type VARCHAR (128) that specifies the SQL schema for
the original XML schema to be updated. The SQL schema is one part of the
SQL identifier used to identify this XML schema in the XSR. (The other part of
the SQL identifier is supplied by the name1 argument.) This argument cannot
have a NULL value. Rules for valid characters and delimiters that apply to any
SQL identifier also apply to this argument.

name1
An input argument of type VARCHAR (128) that specifies the name of the
original XML schema to be updated. The complete SQL identifier for the XML
schema is rschema1.name1. This XML schema must already be registered and
completed in the XSR. This argument cannot have a NULL value. Rules for
valid characters and delimiters that apply to any SQL identifier also apply to
this argument.

rschema2
An input argument of type VARCHAR (128) that specifies the SQL schema for
the new XML schema that will be used to update the original XML schema.
The SQL schema is one part of the SQL identifier used to identify this XML
schema in the XSR. (The other part of the SQL identifier is supplied by the
name2 argument.) This argument cannot have a NULL value. Rules for valid
characters and delimiters that apply to any SQL identifier also apply to this
argument.

name2
An input argument of type VARCHAR (128) that specifies the name of the new
XML schema that will be used to update the original XML schema. The
complete SQL identifier for the XML schema is rschema2.name2. This XML
schema must already be registered and completed in the XSR. This argument
cannot have a NULL value. Rules for valid characters and delimiters that
apply to any SQL identifier also apply to this argument.

dropnewschema
An input parameter of type integer that indicates whether the new XML
schema should be dropped after it is used to update the original XML schema.
Setting this parameter to any nonzero value will cause the new XML schema to
be dropped. This argument cannot have a null value.

Example:
CALL SYSPROC.XSR_UPDATE(
  'STORE',
  'PROD',
  'STORE',
  'NEWPROD',
  1)

The contents of the XML schema STORE.PROD is updated with the contents of
STORE.NEWPROD, and the XML schema STORE.NEWPROD is dropped.
Chapter 5. SQL queries

A query specifies a result table. A query is a component of certain SQL statements. The three forms of a query are:

- subselect
- fullselect
- select-statement.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- For each table or view identified in the query, one of the following:
  - SELECT privilege on the table or view
  - CONTROL privilege on the table or view
- DATAACCESS authority

For each global variable used as an expression in the query, the privileges held by the authorization ID of the statement must include one of the following:

- READ privilege on the global variable that is not defined in a module
- EXECUTE privilege on the module of the global variable that is defined in a module

If the query contains an SQL data change statement, the authorization requirements of that statement also apply to the query.

Group privileges, with the exception of PUBLIC, are not checked for queries that are contained in static SQL statements or DDL statements.

For nicknames, authorization requirements of the data source for the object referenced by the nickname are applied when the query is processed. The authorization ID of the statement may be mapped to a different authorization ID at the data source.

Queries and table expressions

A query is a component of certain SQL statements; it specifies a (temporary) result table.

A table expression creates a temporary result table from a simple query. Clauses further refine the result table. For example, you can use a table expression as a query to select all of the managers from several departments, specify that they must have over 15 years of working experience, and be located at the New York branch office.

A common table expression is like a temporary view within a complex query. It can be referenced in other places within the query, and can be used in place of a view. Each use of a specific common table expression within a complex query shares the same temporary view.
Recursive use of a common table expression within a query can be used to support applications such as airline reservation systems, bill of materials (BOM) generators, and network planning.
The subselect is a component of the fullselect.

A subselect specifies a result table derived from the tables, views or nicknames identified in the FROM clause. The derivation can be described as a sequence of operations in which the result of each operation is input for the next. (This is only a way of describing the subselect. The method used to perform the derivation can be quite different from this description. If portions of the subselect do not actually need to be executed for the correct result to be obtained, they might or might not be executed.)

The authorization for a subselect is described in the Authorization section in "SQL queries".

The clauses of the subselect are processed in the following sequence:
1. FROM clause
2. WHERE clause
3. GROUP BY clause
4. HAVING clause
5. SELECT clause
6. ORDER BY clause
7. FETCH FIRST clause

A subselect that contains an ORDER BY or FETCH FIRST clause cannot be specified:
• In the outermost fullselect of a view.
• In a materialized query table.
• Unless the subselect is enclosed in parenthesis.

For example, the following is not valid (SQLSTATE 428FJ):

```
SELECT * FROM T1
ORDER BY C1
UNION
SELECT * FROM T2
ORDER BY C1
```

The following example is valid:

```
(SELECT * FROM T1
ORDER BY C1)
UNION
(SELECT * FROM T2
ORDER BY C1)
```
**Note:** An ORDER BY clause in a subselect does not affect the order of the rows returned by a query. An ORDER BY clause only affects the order of the rows returned if it is specified in the outermost fullselect.

**select-clause**

The SELECT clause specifies the columns of the final result table, R. The column values are produced by the application of the select list to R. The select list is the names or expressions specified in the SELECT clause, and R is the result of the previous operation of the subselect. For example, if the only clauses specified are SELECT, FROM, and WHERE, R is the result of that WHERE clause.

**ALL**
Retains all rows of the final result table, and does not eliminate redundant duplicates. This is the default.

**DISTINCT**
Eliminates all but one of each set of duplicate rows of the final result table. If DISTINCT is used, no string column of the result table can be a LOB type, distinct type based on LOB, or structured type. DISTINCT may be used more than once in a subselect. This includes SELECT DISTINCT, the use of DISTINCT in an aggregate function of the select list or HAVING clause, and subqueries of the subselect.

Two rows are duplicates of one another only if each value in the first is equal to the corresponding value in the second. For determining duplicates, two null values are considered equal, and two different decimal floating-point representations of the same number are considered equal. For example, -0 is equal to +0 and 2.0 is equal to 2.00. Each of the decimal floating-point special values are also considered equal: -NAN equals -NAN, -SNAN equals -SNAN, -INFINITY equals -INFINITY, INFINITY equals INFINITY, SNAN equals SNAN, and NAN equals NAN.

When the data type of a column is decimal floating-point, and multiple representations of the same number exist in the column, the particular value that is returned for a SELECT DISTINCT can be any one of the representations in the column. For more information, see "Numeric comparisons" on page 123.

For compatibility with other SQL implementations, UNIQUE can be specified as a synonym for DISTINCT.

**Select list notation**

* Represents a list of names that identify the columns of table R, excluding any columns defined as IMPLICITLY HIDDEN. The first name in the list identifies the first column of R, the second name identifies the second column of R, and so on.
The list of names is established when the program containing the SELECT clause is bound. Hence * (the asterisk) does not identify any columns that have been added to a table after the statement containing the table reference has been bound.

expression

Specifies the values of a result column. Can be any expression that is a valid SQL language element, but commonly includes column names. Each column name used in the select list must unambiguously identify a column of R. The result type of the expression cannot be a row type (SQLSTATE 428H2).

d new-column-name or AS new-column-name

Names or renames the result column. The name must not be qualified and does not have to be unique. Subsequent usage of column-name is limited as follows:

- A new-column-name specified in the AS clause can be used in the order-by-clause, provided the name is unique.
- A new-column-name specified in the AS clause of the select list cannot be used in any other clause within the subselect (where-clause, group-by-clause or having-clause).
- A new-column-name specified in the AS clause cannot be used in the update-clause.
- A new-column-name specified in the AS clause is known outside the fullselect of nested table expressions, common table expressions and CREATE VIEW.

name.*

Represents the list of names that identify the columns of the result table identified by exposed-name, excluding any columns defined as IMPLICITLY HIDDEN. The exposed-name may be a table name, view name, nickname, or correlation name, and must designate a table, view or nickname named in the FROM clause. The first name in the list identifies the first column of the table, view or nickname, the second name in the list identifies the second column of the table, view or nickname, and so on.

The list of names is established when the statement containing the SELECT clause is bound. Therefore, * does not identify any columns that have been added to a table after the statement has been bound.

The number of columns in the result of SELECT is the same as the number of expressions in the operational form of the select list (that is, the list established when the statement is prepared), and cannot exceed 500 for a 4K page size or 1012 for an 8K, 16K, or 32K page size.

Limitations on string columns

For limitations on the select list, see “Restrictions Using Varying-Length Character Strings”.

Applying the select list

Some of the results of applying the select list to R depend on whether or not GROUP BY or HAVING is used. The results are described in two separate lists.

If GROUP BY or HAVING is used

- An expression $X$ (not an aggregate function) used in the select list must have a GROUP BY clause with:
– a grouping-expression in which each expression or column-name unambiguously identifies a column of R (see “group-by-clause” on page 705) or
– each column of R referenced in X as a separate grouping-expression.

• The select list is applied to each group of R, and the result contains as many rows as there are groups in R. When the select list is applied to a group of R, that group is the source of the arguments of the aggregate functions in the select list.

If neither GROUP BY nor HAVING is used
• Either the select list must not include any aggregate functions, or each column-name in the select list must be specified within an aggregate function or must be a correlated column reference.
• If the select does not include aggregate functions, then the select list is applied to each row of R and the result contains as many rows as there are rows in R.
• If the select list is a list of aggregate functions, then R is the source of the arguments of the functions and the result of applying the select list is one row.

In either case the nth column of the result contains the values specified by applying the nth expression in the operational form of the select list.

Null attributes of result columns

Result columns do not allow null values if they are derived from:
• A column that does not allow null values
• A constant
• The COUNT or COUNT_BIG function
• A host variable that does not have an indicator variable
• A scalar function or expression that does not include an operand that allows nulls

Result columns allow null values if they are derived from:
• Any aggregate function except COUNT or COUNT_BIG
• A column that allows null values
• A scalar function or expression that includes an operand that allows nulls
• A NULLIF function with arguments containing equal values
• A host variable that has an indicator variable, an SQL parameter, an SQL variable, or a global variable
• A result of a set operation if at least one of the corresponding items in the select list is nullable
• An arithmetic expression or view column that is derived from an arithmetic expression and the database is configured with dft_sqlmathwarn set to Yes
• A scalar subselect
• A dereference operation
• A GROUPING SETS grouping-expression

Names of result columns
• If the AS clause is specified, the name of the result column is the name specified on the AS clause.
• If the AS clause is not specified and a column list is specified in the correlation clause, the name of the result column is the corresponding name in the correlation column list.

• If neither an AS clause nor a column list in the correlation clause is specified and the result column is derived only from a single column (without any functions or operators), then the result column name is the unqualified name of that column.

• If neither an AS clause nor a column list in the correlation clause is specified and the result column is derived only from a single SQL variable or SQL parameter (without any functions or operators), then the result column name is the unqualified name of that SQL variable or SQL parameter.

• If neither an AS clause nor a column list in the correlation clause is specified and the result column is derived using a dereference operation, then the result column name is the unqualified name of the target column of the dereference operation.

• All other result column names are unnamed. The system assigns temporary numbers (as character strings) to these columns.

**Data types of result columns**

Each column of the result of SELECT acquires a data type from the expression from which it is derived.

<table>
<thead>
<tr>
<th>When the expression is ...</th>
<th>The data type of the result column is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>the name of any numeric column</td>
<td>the same as the data type of the column, with the same precision and scale for DECIMAL columns, or the same precision for DECFLOAT columns.</td>
</tr>
<tr>
<td>a constant</td>
<td>the same as the data type of the constant.</td>
</tr>
<tr>
<td>the name of any numeric variable</td>
<td>the same as the data type of the variable, with the same precision and scale for DECIMAL variables, or the same precision for DECFLOAT variables.</td>
</tr>
<tr>
<td>the name of any string column</td>
<td>the same as the data type of the column, with the same length attribute.</td>
</tr>
<tr>
<td>the name of any string variable</td>
<td>the same as the data type of the variable, with the same length attribute; if the data type of the variable is not identical to an SQL data type (for example, a NUL-terminated string in C), the result column is a varying-length string.</td>
</tr>
<tr>
<td>the name of a datetime column</td>
<td>the same as the data type of the column.</td>
</tr>
<tr>
<td>the name of a user-defined type column</td>
<td>the same as the data type of the column.</td>
</tr>
<tr>
<td>the name of a reference type column</td>
<td>the same as the data type of the column.</td>
</tr>
</tbody>
</table>

**from-clause**

The FROM clause specifies an intermediate result table.
If only one table-reference is specified, the intermediate result table is simply the result of that table-reference. If more than one table-reference is specified, the intermediate result table consists of all possible combinations of the rows of the specified table-reference (the Cartesian product). Each row of the result is a row from the first table-reference concatenated with a row from the second table-reference, concatenated in turn with a row from the third, and so on. The number of rows in the result is the product of the number of rows in all the individual table references. For a description of table-reference, see "table-reference."
Notes:

1. An XMLTABLE expression can be part of a table-reference. In this case, subexpressions within the XMLTABLE expression are in-scope of prior range variables in the FROM clause. For more information, see the description of “XMLTABLE”.

2. TABLE can be specified in place of LATERAL.

3. WITH ORDINALITY can be specified only if the argument to the UNNEST table function is one or more ordinary array variables; an associative array variable cannot be specified (SQLSTATE 428HT).

Each table-name, view-name or nickname specified as a table-reference must identify an existing table, view or nickname at the application server or the table-name of a common table expression defined preceding the fullselect containing the table-reference. If the table-name references a typed table, the name denotes the UNION ALL of the table with all its subtables, with only the columns of the table-name. Similarly, if the view-name references a typed view, the name denotes the UNION ALL of the view with all its subviews, with only the columns of the view-name.

The use of ONLY(table-name) or ONLY(view-name) means that the rows of the proper subtables or subviews are not included. If the table-name used with ONLY does not have subtables, then ONLY(table-name) is equivalent to specifying table-name. If the view-name used with ONLY does not have subviews, then ONLY(view-name) is equivalent to specifying view-name.
The use of OUTER(table-name) or OUTER(view-name) represents a virtual table. If the table-name or view-name used with OUTER does not have subtables or subviews, then specifying OUTER is equivalent to not specifying OUTER.

OUTER(table-name) is derived from table-name as follows:

- The columns include the columns of table-name followed by the additional columns introduced by each of its subtables (if any). The additional columns are added on the right, traversing the subtable hierarchy in depth-first order. Subtables that have a common parent are traversed in creation order of their types.
- The rows include all the rows of table-name and all the rows of its subtables. Null values are returned for columns that are not in the subtable for the row.

The previous points also apply to OUTER(view-name), substituting view-name for table-name and subview for subtable.

The use of ONLY or OUTER requires the SELECT privilege on every subtable of table-name or subview of view-name.

Each function-name together with the types of its arguments, specified as a table reference must resolve to an existing table function at the application server.

A fullselect in parentheses is called a nested table expression.

A joined-table specifies an intermediate result set that is the result of one or more join operations. For more information, see “joined-table” on page 703.

The exposed names of all table references should be unique. An exposed name is:

- A correlation-name
- A table-name that is not followed by a correlation-name
- A view-name that is not followed by a correlation-name
- A nickname that is not followed by a correlation-name
- An alias-name that is not followed by a correlation-name

If a correlation-clause does not follow a function-name reference, xmltable-expression, nested table expression, or data-change-table-reference, there is no exposed name for that table reference.

Each correlation-name is defined as a designator of the immediately preceding table-name, view-name, nickname, function-name reference, xmltable-expression, nested table expression, or data-change-table-reference. Any qualified reference to a column must use the exposed name. If the same table name, view or nickname name is specified twice, at least one specification should be followed by a correlation-name. The correlation-name is used to qualify references to the columns of the table, view or nickname. When a correlation-name is specified, column-names can also be specified to give names to the columns of the table reference. If the correlation-clause does not include column-names, the exposed column names are determined as follows:

- Column names of the referenced table, view, or nickname when the table-reference is a table-name, view-name, nickname, or alias-name
- Column names specified in the RETURNS clause of the CREATE FUNCTION statement when the table-reference is a function-name reference
- Column names specified in the COLUMNS clause of the xmltable-expression when the table-reference is an xmltable-expression
Column names exposed by the fullselect when the table-reference is a nested-table-expression

Column names from the target table of the data change statement, along with any defined INCLUDE columns when the table-reference is a data-change-table-reference

In general, collection-derived tables, table functions, and nested table expressions can be specified on any from-clause. Columns from the table functions, nested table expressions, or collection-derived tables can be referenced in the select list and in the rest of the subselect using the correlation name. The scope of this correlation name is the same as correlation names for other tables, views, or nicknames in the FROM clause. A nested table expression can be used:

- In place of a view to avoid creating the view (when general use of the view is not required)
- When the desired result table is based on host variables

A collection-derived-table can be used to convert the elements of an array into values of a column in separate rows. If WITH ORDINALITY is specified, an extra column of data type INTEGER is appended. This column contains the position of the element in the array. The columns can be referenced in the select list and the rest of the subselect by using the names specified for the columns in the correlation-clause. The collection-derived-table clause can only be used in a context where arrays are supported (SQLSTATE 42887). See the “UNNEST table function” for details.

An expression in the select list of a nested table expression that is referenced within, or is the target of, a data change statement within a fullselect is only valid when it does not include:

- A function that reads or modifies SQL data
- A function that is non-deterministic
- A function that has external action
- An OLAP function

If a view is referenced directly in, or as the target of a nested table expression in a data change statement within a FROM clause, the view must either be symmetric (have WITH CHECK OPTION specified) or satisfy the restriction for a WITH CHECK OPTION view.

If the target of a data change statement within a FROM clause is a nested table expression, the modified rows are not requalified, WHERE clause predicates are not re-evaluated, and ORDER BY or FETCH FIRST operations are not redone.

The optional tablesample-clause can be used to obtain a random subset (a sample) of the rows from the specified table-name, rather than the entire contents of that table-name, for this query. This sampling is in addition to any predicates that are specified in the where-clause. Unless the optional REPEATABLE clause is specified, each execution of the query will usually yield a different sample, except in degenerate cases where the table is so small relative to the sample size that any sample must return the same rows. The size of the sample is controlled by the numeric-expression1 in parentheses, representing an approximate percentage (P) of the table to be returned. The method by which the sample is obtained is specified after the TABLESAMPLE keyword, and can be either BERNOUILLI or SYSTEM. For both methods, the exact number of rows in the sample may be different for each
subselect

execution of the query, but on average should be approximately P percent of the table, before any predicates further reduce the number of rows.

The \textit{table-name} must be a stored table. It can be a materialized query table (MQT) name, but not a subselect or table expression for which an MQT has been defined, because there is no guarantee that the database manager will route to the MQT for that subselect.

Semantically, sampling of a table occurs before any other query processing, such as applying predicates or performing joins. Repeated accesses of a sampled table within a single execution of a query (such as in a nested-loop join or a correlated subquery) will return the same sample. More than one table may be sampled in a query.

BERNOULLI sampling considers each row individually. It includes each row in the sample with probability P/100 (where P is the value of \textit{numeric-expression1}), and excludes each row with probability 1 - P/100, independently of the other rows. So if the \textit{numeric-expression1} evaluated to the value 10, representing a ten percent sample, each row would be included with probability 0.1, and excluded with probability 0.9.

SYSTEM sampling permits the database manager to determine the most efficient manner in which to perform the sampling. In most cases, SYSTEM sampling applied to a \textit{table-name} means that each page of \textit{table-name} is included in the sample with probability P/100, and excluded with probability 1 - P/100. All rows on each page that is included qualify for the sample. SYSTEM sampling of a \textit{table-name} generally executes much faster than BERNOUlli sampling, because fewer data pages need to be retrieved; however, it can often yield less accurate estimates for aggregate functions (SUM(SALES), for example), especially if the rows of \textit{table-name} are clustered on any columns referenced in that query. The optimizer may in certain circumstances decide that it is more efficient to perform SYSTEM sampling as if it were BERNOUlli sampling, for example when a predicate on \textit{table-name} can be applied by an index and is much more selective than the sampling rate P.

The \textit{numeric-expression1} specifies the size of the sample to be obtained from \textit{table-name}, expressed as a percentage. It must be a constant numeric expression that cannot contain columns. The expression must evaluate to a positive number that is less than or equal to 100, but can be between 1 and 0. For example, a value of 0.01 represents one one-hundredth of a percent, meaning that 1 row in 10 000 would be sampled, on average. A \textit{numeric-expression1} that evaluates to 100 is handled as if the tablesample-clause were not specified. If \textit{numeric-expression1} evaluates to the null value, or to a value that is greater than 100 or less than 0, an error is returned (SQLSTATE 2202H).

It is sometimes desirable for sampling to be repeatable from one execution of the query to the next; for example, during regression testing or query "debugging". This can be accomplished by specifying the \textbf{REPEATABLE} clause. The \textbf{REPEATABLE} clause requires the specification of a \textit{numeric-expression2} in parentheses, which serves the same role as the seed in a random number generator. Adding the \textbf{REPEATABLE} clause to the tablesample-clause of any \textit{table-name} ensures that repeated executions of that query (using the same value for \textit{numeric-expression2}) return the same sample, assuming, of course, that the data itself has not been updated, reorganized, or repartitioned. To guarantee that the same sample of \textit{table-name} is used across multiple queries, use of a global
temporary table is recommended. Alternatively, the multiple queries could be combined into one query, with multiple references to a sample that is defined using the WITH clause.

Following are some examples:

*Example 1:* Request a 10% Bernoulli sample of the Sales table for auditing purposes.

```
SELECT * FROM Sales
TABLESAMPLE BERNOULLI(10)
```

*Example 2:* Compute the total sales revenue in the Northeast region for each product category, using a random 1% SYSTEM sample of the Sales table. The semantics of SUM are for the sample itself, so to extrapolate the sales to the entire Sales table, the query must divide that SUM by the sampling rate (0.01).

```
SELECT SUM(Sales.Revenue) / (0.01)
FROM Sales TABLESAMPLE SYSTEM(1)
WHERE Sales.RegionName = 'Northeast'
GROUP BY Sales.ProductCategory
```

*Example 3:* Using the REPEATABLE clause, modify the previous query to ensure that the same (yet random) result is obtained each time the query is executed. (The value of the constant enclosed by parentheses is arbitrary.)

```
SELECT SUM(Sales.Revenue) / (0.01)
FROM Sales TABLESAMPLE SYSTEM(1) REPEATABLE(3578231)
WHERE Sales.RegionName = 'Northeast'
GROUP BY Sales.ProductCategory
```

**Table function references**

In general, a table function, together with its argument values, can be referenced in the FROM clause of a SELECT in exactly the same way as a table or view. There are, however, some special considerations which apply.

- **Table Function Column Names**
  
  Unless alternate column names are provided following the *correlation-name*, the column names for the table function are those specified in the RETURNS clause of the CREATE FUNCTION statement. This is analogous to the names of the columns of a table, which are defined in the CREATE TABLE statement.

- **Table Function Resolution**
  
  The arguments specified in a table function reference, together with the function name, are used by an algorithm called *function resolution* to determine the exact function to be used. This is no different from what happens with other functions (such as scalar functions) that are used in a statement.

- **Table Function Arguments**
  
  As with scalar function arguments, table function arguments can in general be any valid SQL expression. The following examples are valid syntax:

```
Example 1: SELECT c1
           FROM TABLE( tf1('Zachary') ) AS z
           WHERE c2 = 'FLORIDA';
```

```
Example 2: SELECT c1
           FROM TABLE( tf2 (:hostvar1, CURRENT DATE) ) AS z;
```

```
Example 3: SELECT c1
           FROM t
           WHERE c2 IN
```

subselect
**Table Functions That Modify SQL Data**

Table functions that are specified with the MODIFIES SQL DATA option can only be used as the last table reference in a **select-statement**, **common-table-expression**, or RETURN statement that is a subselect, a SELECT INTO, or a **row-fullselect** in a SET statement. Only one table function is allowed in one FROM clause, and the table function arguments must be correlated to all other table references in the subselect (SQLSTATE 429BL). The following examples have valid syntax for a table function with the MODIFIES SQL DATA property:

**Example 1:**
```
SELECT c1 FROM TABLE( tfmod('Jones') ) AS z
```

**Example 2:**
```
SELECT c1 FROM t1, t2, TABLE( tfmod(t1.c1, t2.c1) ) AS z
```

**Example 3:**
```
SET var = (SELECT c1 FROM TABLE( tfmod('Jones') ) AS z
```

**Example 4:**
```
RETURN SELECT c1 FROM TABLE( tfmod('Jones') ) AS z
```

**Example 5:**
```
WITH v1 AS (SELECT c1 FROM TABLE( tfmod(:hostvar1) ) AS z,
SELECT c1 FROM v1, t1 WHERE v1.c1 = t1.c1
```

**Error tolerant nested-table-expression**

Certain errors that occur within a **nested-table-expression** can be tolerated, and instead of returning an error, the query can continue and return a result.

Specifying the RETURN DATA UNTIL clause will cause any rows that are returned from the fullselect before the indicated condition is encountered to make up the result set from the fullselect. This means that a partial result set (which could also be an empty result set) from the fullselect is acceptable as the result for the **nested-table-expression**.

The FEDERATED keyword restricts the condition to handle only errors that occur at a remote data source.

The condition can be specified as an SQLSTATE value, with a **string-constant** length of 5. You can optionally specify an SQLCODE value for each specified SQLSTATE value. For portable applications, specify SQLSTATE values as much as possible, because SQLCODE values are generally not portable across platforms and are not part of the SQL standard.

Only certain conditions can be tolerated. Errors that do not allow the rest of the query to be executed cannot be tolerated, and an error is returned for the whole query. The **specific-condition-value** might specify conditions that cannot actually be tolerated by the database manager, even if a specific SQLSTATE or SQLCODE value is specified, and for these cases, an error is returned.

The following SQLSTATE values and SQLCODE values have the potential, when specified, to be tolerated by the database manager:
A query or view containing an error tolerant nested-table-expression is read-only.

The fullselect of an error tolerant nested-table-expression is not optimized using materialized query tables.

**Correlated references in table-references**

Correlated references can be used in nested table expressions or as arguments to table functions. The basic rule that applies for both these cases is that the correlated reference must be from a table-reference at a higher level in the hierarchy of subqueries. This hierarchy includes the table-references that have already been resolved in the left-to-right processing of the FROM clause. For nested table expressions, the LATERAL keyword must appear before the fullselect. So the following examples are valid syntax:

**Example 1:**
```
SELECT t.c1, z.c5
FROM t, TABLE(tf3(t.c2)) AS z
WHERE t.c3 = z.c4;
```

**Example 2:**
```
SELECT t.c1, z.c5
FROM t, TABLE(tf4(2 * t.c2)) AS z
WHERE t.c3 = z.c4;
```

**Example 3:**
```
SELECT d.deptno, d.deptname,
empinfo.avgsal, empinfo.empcount
FROM department d,
LATERAL (SELECT AVG(e.salary)
AS avgsal,
COUNT(*) AS empcount
FROM employee e
WHERE e.workdept=d.deptno)
AS empinfo;
```

But the following examples are not valid:

**Example 4:**
```
SELECT t.c1, z.c5
FROM TABLE(tf6(t.c2)) AS z, t
WHERE t.c3 = z.c4;
```

**Example 5:**
```
SELECT a.c1, b.c5
FROM TABLE(tf7a(b.c2)) AS a, TABLE(tf7b(a.c6)) AS b
WHERE a.c3 = b.c4;
```

**Example 6:**
```
SELECT d.deptno, d.deptname,
empinfo.avgsal, empinfo.empcount
FROM department d,
(SELECT AVG(e.salary)
AS avgsal,
COUNT(*) AS empcount
FROM employee e
WHERE e.workdept=d.deptno)
AS empinfo;
```
Data change table references

A `data-change-table-reference` clause specifies an intermediate result table. This table is based on the rows that are directly changed by the searched UPDATE, searched DELETE, or INSERT statement that is included in the clause. A `data-change-table-reference` can be specified as the only `table-reference` in the FROM clause of the outer fullselect that is used in a `select-statement`, a SELECT INTO statement, or a common table expression. A `data-change-table-reference` can be specified as the only table reference in the only fullselect in a SET Variable statement (SQLSTATE 428FL). The target table or view of the data change statement is considered to be a table or view that is referenced in the query; therefore, the authorization ID of the query must have SELECT privilege on that target table or view. A `data-change-table-reference` clause cannot be specified in a view definition, materialized query table definition, or FOR statement (SQLSTATE 428FL).

The target of the UPDATE, DELETE, or INSERT statement cannot be a temporary view defined in a common table expression (SQLSTATE 42807).

**FINAL TABLE**
- Specifies that the rows of the intermediate result table represent the set of rows that are changed by the SQL data change statement as they appear at the completion of the data change statement. If there are AFTER triggers or referential constraints that result in further operations on the table that is the target of the SQL data change statement, an error is returned (SQLSTATE 560C6). If the target of the SQL data change statement is a view that is defined with an INSTEAD OF trigger for the type of data change, an error is returned (SQLSTATE 428G3).

**NEW TABLE**
- Specifies that the rows of the intermediate result table represent the set of rows that are changed by the SQL data change statement prior to the application of referential constraints and AFTER triggers. Data in the target table at the completion of the statement might not match the data in the intermediate result table because of additional processing for referential constraints and AFTER triggers.

**OLD TABLE**
- Specifies that the rows of the intermediate result table represent the set of rows that are changed by the SQL data change statement as they existed prior to the application of the data change statement.

*(searched-update-statement)*
- Specifies a searched UPDATE statement. A WHERE clause or a SET clause in the UPDATE statement cannot contain correlated references to columns outside of the UPDATE statement.

*(searched-delete-statement)*
- Specifies a searched DELETE statement. A WHERE clause in the DELETE statement cannot contain correlated references to columns outside of the DELETE statement.

*(insert-statement)*
- Specifies an INSERT statement. A fullselect in the INSERT statement cannot contain correlated references to columns outside of the fullselect of the INSERT statement.

The content of the intermediate result table for a `data-change-table-reference` is determined when the cursor opens. The intermediate result table contains all...
manipulated rows, including all the columns in the specified target table or view. All the columns of the target table or view for an SQL data change statement are accessible using the column names from the target table or view. If an INCLUDE clause was specified within a data change statement, the intermediate result table will contain these additional columns.

**joined-table**

A *joined table* specifies an intermediate result table that is the result of either an inner join or an outer join. The table is derived by applying one of the join operators: CROSS, INNER, LEFT OUTER, RIGHT OUTER, or FULL OUTER to its operands.

Cross joins represent the cross product of the tables, where each row of the left table is combined with every row of the right table. Inner joins can be thought of as the cross product of the tables, keeping only the rows where the join condition is true. The result table might be missing rows from either or both of the joined tables. Outer joins include the inner join and preserve these missing rows. There are three types of outer joins:

- **Left outer join** includes rows from the left table that were missing from the inner join.
- **Right outer join** includes rows from the right table that were missing from the inner join.
- **Full outer join** includes rows from both the left and right tables that were missing from the inner join.

If a join-operator is not specified, INNER is implicit. The order in which multiple joins are performed can affect the result. Joins can be nested within other joins. The order of processing for joins is generally from left to right, but based on the position of the required join-condition. Parentheses are recommended to make the order of nested joins more readable. For example:

```sql
TB1 LEFT JOIN TB2 ON TB1.C1=TB2.C1
  RIGHT JOIN TB3 LEFT JOIN TB4 ON TB3.C1=TB4.C1
    ON TB1.C1=TB3.C1
```

is the same as:

```sql
(TB1 LEFT JOIN TB2 ON TB1.C1=TB2.C1)
  RIGHT JOIN (TB3 LEFT JOIN TB4 ON TB3.C1=TB4.C1)
    ON TB1.C1=TB3.C1
```
A joined table can be used in any context in which any form of the SELECT statement is used. A view or a cursor is read-only if its SELECT statement includes a joined table.

A join-condition is a search-condition, except that:
- It cannot contain any subqueries, scalar or otherwise
- It cannot include any dereference operations or the DEREF function, where the reference value is other than the object identifier column
- It cannot include an SQL function
- Any column referenced in an expression of the join-condition must be a column of one of the operand tables of the associated join (in the scope of the same joined-table clause)
- Any function referenced in an expression of the join-condition of a full outer join must be deterministic and have no external action
- It cannot include an XMLQUERY or XMLEXISTS expression

An error occurs if the join condition does not comply with these rules (SQLSTATE 42972).

Column references are resolved using the rules for resolution of column name qualifiers. The same rules that apply to predicates apply to join conditions.

**Join operations**

A join-condition specifies pairings of T1 and T2, where T1 and T2 are the left and right operand tables of the JOIN operator of the join-condition. For all possible combinations of rows of T1 and T2, a row of T1 is paired with a row of T2 if the join-condition is true. When a row of T1 is joined with a row of T2, a row in the result consists of the values of that row of T1 concatenated with the values of that row of T2. The execution might involve the generation of a null row. The null row of a table consists of a null value for each column of the table, regardless of whether the columns allow null values.

The following summarizes the result of the join operations:
- The result of T1 CROSS JOIN T2 consists of all possible pairings of their rows.
- The result of T1 INNER JOIN T2 consists of their paired rows where the join-condition is true.
- The result of T1 LEFT OUTER JOIN T2 consists of their paired rows where the join-condition is true and, for each unpaired row of T1, the concatenation of that row with the null row of T2. All columns derived from T2 allow null values.
- The result of T1 RIGHT OUTER JOIN T2 consists of their paired rows where the join-condition is true and, for each unpaired row of T2, the concatenation of that row with the null row of T1. All columns derived from T1 allow null values.
- The result of T1 FULL OUTER JOIN T2 consists of their paired rows and, for each unpaired row of T2, the concatenation of that row with the null row of T1 and, for each unpaired row of T1, the concatenation of that row with the null row of T2. All columns derived from T1 and T2 allow null values.

**where-clause**

```sql
WHERE search-condition
```
The WHERE clause specifies an intermediate result table that consists of those rows of R for which the search-condition is true. R is the result of the FROM clause of the subselect.

The search-condition must conform to the following rules:

- Each column-name must unambiguously identify a column of R or be a correlated reference. A column-name is a correlated reference if it identifies a column of a table-reference in an outer subselect.
- An aggregate function must not be specified unless the WHERE clause is specified in a subquery of a HAVING clause and the argument of the function is a correlated reference to a group.

Any subquery in the search-condition is effectively executed for each row of R, and the results are used in the application of the search-condition to the given row of R. A subquery is actually executed for each row of R only if it includes a correlated reference. In fact, a subquery with no correlated references may be executed just once, whereas a subquery with a correlated reference may have to be executed once for each row.

The GROUP BY clause specifies an intermediate result table that consists of a grouping of the rows of R. R is the result of the previous clause of the subselect.

In its simplest form, a GROUP BY clause contains a grouping expression. A grouping expression is an expression used in defining the grouping of R. Each expression or column name included in grouping-expression must unambiguously identify a column of R (SQLSTATE 42702 or 42703). A grouping expression cannot include a scalar fullselect or an XMLQUERY or XMLEXISTS expression (SQLSTATE 42822), or any expression or function that is not deterministic or has an external action (SQLSTATE 42845).

Note: The following expressions, which do not contain an explicit column reference, can be used in a grouping-expression to identify a column of R:

- ROW CHANGE TIMESTAMP FOR table-designator
- ROW CHANGE TOKEN FOR table-designator
- RID_BIT or RID scalar function

More complex forms of the GROUP BY clause include grouping-sets and super-groups. For a description of these forms, see "grouping-sets" on page 706 and "super-groups" on page 707, respectively.

The result of GROUP BY is a set of groups of rows. Each row in this result represents the set of rows for which the grouping-expression is equal. For grouping, all null values from a grouping-expression are considered equal.
If a grouping-expression contains decimal floating-point columns, and multiple representations of the same number exist in these columns, the number that is returned can be any of the representations of the number.

A grouping-expression can be used in a search condition in a HAVING clause, in an expression in a SELECT clause or in a sort-key-expression of an ORDER BY clause (see "order-by-clause" on page 712 for details). In each case, the reference specifies only one value for each group. For example, if the grouping-expression is col1+col2, then an allowed expression in the select list would be col1+col2+3. Associativity rules for expressions would disallow the similar expression, 3+col1+col2, unless parentheses are used to ensure that the corresponding expression is evaluated in the same order. Thus, 3+(col1+col2) would also be allowed in the select list. If the concatenation operator is used, the grouping-expression must be used exactly as the expression was specified in the select list.

If the grouping-expression contains varying-length strings with trailing blanks, the values in the group can differ in the number of trailing blanks and may not all have the same length. In that case, a reference to the grouping-expression still specifies only one value for each group, but the value for a group is chosen arbitrarily from the available set of values. Thus, the actual length of the result value is unpredictable.

As noted, there are some cases where the GROUP BY clause cannot refer directly to a column that is specified in the SELECT clause as an expression (scalar-fullselect, not deterministic or external action functions). To group using such an expression, use a nested table expression or a common table expression to first provide a result table with the expression as a column of the result. For an example using nested table expressions, see Example A9.

A grouping-sets specification allows multiple grouping clauses to be specified in a single statement. This can be thought of as the union of two or more groups of rows into a single result set. It is logically equivalent to the union of multiple subselects with the group by clause in each subselect corresponding to one grouping set. A grouping set can be a single element or can be a list of elements delimited by parentheses, where an element is either a grouping-expression or a super-group. Using grouping-sets allows the groups to be computed with a single pass over the base table.

The grouping-sets specification allows either a simple grouping-expression to be used, or the more complex forms of super-groups. For a description of super-groups, see "super-groups" on page 707.
Note that grouping sets are the fundamental building blocks for GROUP BY operations. A simple GROUP BY with a single column can be considered a grouping set with one element. For example:

```
GROUP BY a
```

is the same as

```
GROUP BY GROUPING SETS((a))
```

and

```
GROUP BY a,b,c
```

is the same as

```
GROUP BY GROUPING SETS((a,b,c))
```

Non-aggregation columns from the select list of the subselect that are excluded from a grouping set will return a null for such columns for each row generated for that grouping set. This reflects the fact that aggregation was done without considering the values for those columns.

Example C2 through Example C7 illustrate the use of grouping sets.

**super-groups**

\[
\text{ROLLUP} \left( \text{grouping-expression-list} \right)
\]

\[
\text{CUBE} \left( \text{grouping-expression-list} \right)
\]

\[
\text{grand-total}
\]

**Notes:**

1. Alternate specification when used alone in group-by-clause is: grouping-expression-list WITH ROLLUP.
2. Alternate specification when used alone in group-by-clause is: grouping-expression-list WITH CUBE.

ROLLUP \( \text{grouping-expression-list} \)

A ROLLUP grouping is an extension to the GROUP BY clause that produces a result set containing sub-total rows in addition to the “regular” grouped rows. Sub-total rows are “super-aggregate” rows that contain further aggregates whose values are derived by applying the same aggregate functions that were
used to obtain the grouped rows. These rows are called sub-total rows, because that is their most common use; however, any aggregate function can be used for the aggregation. For instance, MAX and AVG are used in Example C8. The GROUPING aggregate function can be used to indicate if a row was generated by the super-group.

A ROLLUP grouping is a series of grouping-sets. The general specification of a ROLLUP with \( n \) elements

\[
\text{GROUP BY ROLLUP}(c_1, c_2, \ldots, c_{n-1}, c_n)
\]

is equivalent to

\[
\text{GROUP BY GROUPING SETS}((c_1, c_2, \ldots, c_n),
   (c_1, c_2, \ldots, c_{n-1}),
   \ldots,
   (c_1),
   (c_n),
   (\) )
\]

Note that the \( n \) elements of the ROLLUP translate to \( n+1 \) grouping sets. Note also that the order in which the grouping-expressions is specified is significant for ROLLUP. For example:

\[
\text{GROUP BY ROLLUP}(a,b)
\]

is equivalent to

\[
\text{GROUP BY GROUPING SETS}((a,b),
   (a),
   (b),
   (\) )
\]

while

\[
\text{GROUP BY ROLLUP}(b,a)
\]

is the same as

\[
\text{GROUP BY GROUPING SETS}((b,a),
   (b),
   (a),
   (\) )
\]

The ORDER BY clause is the only way to guarantee the order of the rows in the result set. Example C3 illustrates the use of ROLLUP.

CUBE ( grouping-expression-list )

A CUBE grouping is an extension to the GROUP BY clause that produces a result set that contains all the rows of a ROLLUP aggregation and, in addition, contains "cross-tabulation" rows. Cross-tabulation rows are additional "super-aggregate" rows that are not part of an aggregation with sub-totals. The GROUPING aggregate function can be used to indicate if a row was generated by the super-group.

Like a ROLLUP, a CUBE grouping can also be thought of as a series of grouping-sets. In the case of a CUBE, all permutations of the cubed grouping-expression-list are computed along with the grand total. Therefore, the \( n \) elements of a CUBE translate to \( 2^n \) (2 to the power \( n \)) grouping-sets. For example, a specification of:

\[
\text{GROUP BY CUBE}(a,b,c)
\]

is equivalent to:

\[
\text{GROUP BY GROUPING SETS}((a,b,c),
   (a,b),
   (a,c),
   (b,c)
   (\) )
\]
Note that the three elements of the CUBE translate into eight grouping sets. The order of specification of elements does not matter for CUBE. 'CUBE (DayOfYear, Sales_Person)' and 'CUBE (Sales_Person, DayOfYear)' yield the same result sets. The use of the word 'same' applies to content of the result set, not to its order. The ORDER BY clause is the only way to guarantee the order of the rows in the result set. Example C4 illustrates the use of CUBE.

A grouping-expression-list is used within a CUBE or ROLLUP clause to define the number of elements in the CUBE or ROLLUP operation. This is controlled by using parentheses to delimit elements with multiple grouping-expressions.

The rules for a grouping-expression are described in “group-by-clause” on page 705. For example, suppose that a query is to return the total expenses for the ROLLUP of City within a Province but not within a County. However, the clause:

GROUP BY ROLLUP(Province, County, City)

results in unwanted subtotal rows for the County. In the clause:

GROUP BY ROLLUP(Province, (County, City))

the composite (County, City) forms one element in the ROLLUP and, therefore, a query that uses this clause will yield the desired result. In other words, the two-element ROLLUP:

GROUP BY ROLLUP(Province, (County, City))

generates:

GROUP BY GROUPING SETS((Province, County, City)
    (Province)
    ()
    )

and the three-element ROLLUP generates:

GROUP BY GROUPING SETS((Province, County, City)
    (Province, County)
    (Province)
    ()
    )

Example C2 also utilizes composite column values.

Both CUBE and ROLLUP return a row which is the overall (grand total) aggregation. This may be separately specified with empty parentheses within the GROUPING SET clause. It may also be specified directly in the GROUP BY clause, although there is no effect on the result of the query. Example C4 uses the grand-total syntax.

Combining grouping sets

This can be used to combine any of the types of GROUP BY clauses. When simple grouping-expression fields are combined with other groups, they are “appended” to the beginning of the resulting grouping sets. When ROLLUP or CUBE expressions
are combined, they operate like "multipliers" on the remaining expression, forming additional grouping set entries according to the definition of either ROLLUP or CUBE.

For instance, combining grouping-expression elements acts as follows:

GROUP BY a, ROLLUP(b,c)

is equivalent to

GROUP BY GROUPING SETS((a,b,c)
  (a,b)
  (a) )

Or similarly,

GROUP BY a, b, ROLLUP(c,d)

is equivalent to

GROUP BY GROUPING SETS((a,b,c,d)
  (a,b,c)
  (a,b) )

Combining of ROLLUP elements acts as follows:

GROUP BY ROLLUP(a), ROLLUP(b,c)

is equivalent to

GROUP BY GROUPING SETS((a,b,c)
  (a,b)
  (a)
  (b,c)
  (b)
  (c)
  () )

Similarly,

GROUP BY ROLLUP(a), CUBE(b,c)

is equivalent to

GROUP BY GROUPING SETS((a,b,c)
  (a,b)
  (a,c)
  (a)
  (b,c)
  (b)
  (c)
  (c) )

Combining of CUBE and ROLLUP elements acts as follows:

GROUP BY CUBE(a,b), ROLLUP(c,d)

is equivalent to

GROUP BY GROUPING SETS((a,b,c,d)
  (a,b,c)
  (a,b)
  (a,c,d)
  (a,c)
  (a)
  (b,c,d)
  (b,c)
Like a simple `grouping-expression`, combining grouping sets also eliminates duplicates within each grouping set. For instance,

```sql
GROUP BY a, ROLLUP(a,b)
```

is equivalent to

```sql
GROUP BY GROUPING SETS((a,b),
(a))
```

A more complete example of combining grouping sets is to construct a result set that eliminates certain rows that would be returned for a full CUBE aggregation.

For example, consider the following GROUP BY clause:

```sql
GROUP BY Region,
  ROLLUP(Sales_Person, WEEK(Sales_Date)),
  CUBE(YEAR(Sales_Date), MONTH(Sales_Date))
```

The column listed immediately to the right of GROUP BY is simply grouped, those within the parenthesis following ROLLUP are rolled up, and those within the parenthesis following CUBE are cubed. Thus, the above clause results in a cube of MONTH within YEAR which is then rolled up within WEEK within Sales_Person within the Region aggregation. It does not result in any grand total row or any cross-tabulation rows on Region, Sales_Person or WEEK(Sales_Date) so produces fewer rows than the clause:

```sql
GROUP BY ROLLUP (Region, Sales_Person, WEEK(Sales_Date),
  YEAR(Sales_Date), MONTH(Sales_Date))
```

### having-clause

The HAVING clause specifies an intermediate result table that consists of those groups of R for which the `search-condition` is true. R is the result of the previous clause of the subselect. If this clause is not GROUP BY, R is considered to be a single group with no grouping columns.

Each `column-name` in the search condition must do one of the following:
- Unambiguously identify a grouping column of R.
- Be specified within an aggregate function.
- Be a correlated reference. A `column-name` is a correlated reference if it identifies a column of a `table-reference` in an outer subselect.

A group of R to which the search condition is applied supplies the argument for each aggregate function in the search condition, except for any function whose argument is a correlated reference.

If the search condition contains a subquery, the subquery can be thought of as being executed each time the search condition is applied to a group of R, and the results used in applying the search condition. In actuality, the subquery is executed for each group only if it contains a correlated reference. For an illustration of the difference, see Example A6 and Example A7.
A correlated reference to a group of R must either identify a grouping column or be contained within an aggregate function.

When HAVING is used without GROUP BY, the select list can only include column names when they are arguments to an aggregate function, correlated column references, global variables, host variables, literals, special registers, SQL variables, or SQL parameters.

Note: The following expressions can only be specified in a HAVING clause if they are contained within an aggregate function (SQLSTATE 42803):
- ROW CHANGE TIMESTAMP FOR table-designator
- ROW CHANGE TOKEN FOR table-designator
- RID_BIT or RID scalar function

**order-by-clause**

```
ORDER BY

ORDER OF table-designator

ORDER BY

sort-key

ASC

DESC

ORDER OF table-designator

order-by-clause
```

**sort-key:**

```
simple-column-name

simple-integer

sort-key-expression
```

The ORDER BY clause specifies an ordering of the rows of the result table. If a single sort specification (one sort-key with associated direction) is identified, the rows are ordered by the values of that sort specification. If more than one sort specification is identified, the rows are ordered by the values of the first identified sort specification, then by the values of the second identified sort specification, and so on. Each sort-key cannot have a data type of CLOB, DBCLOB, BLOB, distinct type on any of these types, or structured type (SQLSTATE 42907).

A named column in the select list may be identified by a sort-key that is a simple-integer or a simple-column-name. An unnamed column in the select list must be identified by an simple-integer or, in some cases, by a sort-key-expression that matches the expression in the select list (see details of sort-key-expression). A column is unnamed if the AS clause is not specified and it is derived from a constant, an expression with operators, or a function.

Ordering is performed in accordance with comparison rules. If an ORDER BY clause contains decimal floating-point columns, and multiple representations of the same number exist in these columns, the ordering of the multiple representations of the same number is unspecified. The null value is higher than all other values. If the ORDER BY clause does not completely order the rows, rows with duplicate values of all identified columns are displayed in an arbitrary order.

**simple-column-name**

Usually identifies a column of the result table. In this case, simple-column-name must be the column name of a named column in the select list.
The *simple-column-name* can also identify a column name of a table, view, or nested table identified in the FROM clause if the query is a subselect. This includes columns defined as implicitly hidden. An error occurs if the subselect:

- Specifies DISTINCT in the select-clause (SQLSTATE 42822)
- Produces a grouped result and the *simple-column-name* is not a grouping-expression (SQLSTATE 42803).

Determining which column is used for ordering the result is described under “Column names in sort keys” below.

**simple-integer**

Must be greater than 0 and not greater than the number of columns in the result table (SQLSTATE 42805). The integer $n$ identifies the $n$th column of the result table.

**sort-key-expression**

An expression that is not simply a column name or an unsigned integer constant. The query to which ordering is applied must be a subselect to use this form of sort-key. The sort-key-expression cannot include a correlated scalar fullselect (SQLSTATE 42703), an XMLQUERY or XMLDOCUMENT expression (SQLSTATE 42822), or a function with an external action (SQLSTATE 42845).

Any *column-name* within a sort-key-expression must conform to the rules described under “Column names in sort keys” below.

There are a number of special cases that further restrict the expressions that can be specified.

- DISTINCT is specified in the SELECT clause of the subselect (SQLSTATE 42822).
  The sort-key-expression must match exactly with an expression in the select list of the subselect (scalar-fullselects are never matched).
- The subselect is grouped (SQLSTATE 42803).
  The sort-key-expression can:
  - be an expression in the select list of the subselect,
  - include a grouping-expression from the GROUP BY clause of the subselect
  - include an aggregate function, constant or host variable.

**ASC**

Uses the values of the column in ascending order. This is the default.

**DESC**

Uses the values of the column in descending order.

**ORDER OF** *table-designator*

Specifies that the same ordering used in *table-designator* should be applied to the result table of the subselect. There must be a table reference matching *table-designator* in the FROM clause of the subselect that specifies this clause (SQLSTATE 42703). The subselect (or fullselect) corresponding to the specified *table-designator* must include an ORDER BY clause that is dependant on the data (SQLSTATE 428FI). The ordering that is applied is the same as if the columns of the ORDER BY clause in the nested subselect (or fullselect) were included in the outer subselect (or fullselect), and these columns were specified in place of the ORDER OF clause.

Note that this form is not allowed in a fullselect (other than the degenerative form of a fullselect). For example, the following is not valid:
subselect

(SELECT C1 FROM T1
 ORDER BY C1)
UNION
SELECT C1 FROM T2
 ORDER BY ORDER OF T1

The following example is valid:
SELECT C1 FROM
 (SELECT C1 FROM T1
  UNION
  SELECT C1 FROM T2
  ORDER BY C1 ) AS UTABLE
ORDER BY ORDER OF UTABLE

INPUT SEQUENCE
Specifies that, for an INSERT statement, the result table will reflect the input order of ordered data rows. INPUT SEQUENCE ordering can only be specified if an INSERT statement is used in a FROM clause (SQLSTATE 428G4). See “table-reference” on page 694. If INPUT SEQUENCE is specified and the input data is not ordered, the INPUT SEQUENCE clause is ignored.

Notes
• Column names in sort keys:
  – The column name is qualified.
    The query must be a subselect (SQLSTATE 42877). The column name must unambiguously identify a column of some table, view or nested table in the FROM clause of the subselect (SQLSTATE 42702). The value of the column is used to compute the value of the sort specification.
  – The column name is unqualified.
    – The query is a subselect.
      If the column name is identical to the name of more than one column of the result table, the column name must unambiguously identify a column of some table, view or nested table in the FROM clause of the ordering subselect (SQLSTATE 42702). If the column name is identical to one column, that column is used to compute the value of the sort specification. If the column name is not identical to a column of the result table, then it must unambiguously identify a column of some table, view or nested table in the FROM clause of the fullselect in the select-statement (SQLSTATE 42702).
    – The query is not a subselect (it includes set operations such as union, except or intersect).
      The column name must not be identical to the name of more than one column of the result table (SQLSTATE 42702). The column name must be identical to exactly one column of the result table (SQLSTATE 42707), and this column is used to compute the value of the sort specification.

• Limits: The use of a sort-key-expression or a simple-column-name where the column is not in the select list may result in the addition of the column or expression to the temporary table used for sorting. This may result in reaching the limit of the number of columns in a table or the limit on the size of a row in a table. Exceeding these limits will result in an error if a temporary table is required to perform the sorting operation.

fetch-first-clause
The fetch-first-clause sets a maximum number of rows that can be retrieved. It lets the database manager know that the application does not want to retrieve more than integer rows, regardless of how many rows there might be in the result table when this clause is not specified. An attempt to fetch beyond integer rows is handled the same way as normal end of data (SQLSTATE 02000). The value of integer must be a positive integer (not zero).

Use of the fetch-first-clause influences query optimization of the subselect or fullselect, based on the fact that at most integer rows will be retrieved. If both the fetch-first-clause is specified in the outermost fullselect and the optimize-for-clause is specified for the select statement, the database manager will use the integer from the optimize-for-clause to influence query optimization of the outermost fullselect.

Limiting the result table to the first integer rows can improve performance. The database manager will cease processing the query once it has determined the first integer rows. If both the fetch-first-clause and the optimize-for-clause are specified, the lower of the integer values from these clauses is used to influence the communications buffer size.

If the fullselect contains an SQL data change statement in the FROM clause, all the rows are modified regardless of the limit on the number of rows to fetch.

**isolation-clause**

The optional isolation-clause specifies the isolation level at which the subselect or fullselect is executed, and whether a specific type of lock is to be acquired.

- RR - Repeatable-Read
- RS - Read Stability
- CS - Cursor Stability
- UR - Uncommitted Read

**lock-request-clause**

The lock-request-clause applies only to queries and to positioning read operations within an insert, update, or delete operation. The insert, update, and delete operations themselves will execute using locking determined by the database manager.
The optional lock-request-clause specifies the type of lock that the database manager is to acquire and hold:

**SHARE**
Concurrent processes can acquire SHARE or UPDATE locks on the data.

**UPDATE**
Concurrent processes can acquire SHARE locks on the data, but no concurrent process can acquire an UPDATE or EXCLUSIVE lock.

**EXCLUSIVE**
Concurrent processes cannot acquire a lock on the data.

**isolation-clause restrictions:**
- The isolation-clause is not supported for a CREATE TABLE, CREATE VIEW, or ALTER TABLE statement (SQLSTATE 42601).
- The isolation-clause cannot be specified for a subselect or fullselect that will cause trigger invocation, referential integrity scans, or MQT maintenance (SQLSTATE 42601).
- A subselect or fullselect cannot include a lock-request-clause if that subselect or fullselect references any SQL functions that are not declared with the option INHERIT ISOLATION LEVEL WITH LOCK REQUEST (SQLSTATE 42601).
- A subselect or fullselect that contains a lock-request-clause are not be eligible for MQT routing.
- If an isolation-clause is specified for a subselect or fullselect within the body of an SQL function, SQL method, or trigger, the clause is ignored and a warning is returned.
- If an isolation-clause is specified for a subselect or fullselect that is used by a scrollable cursor, the clause is ignored and a warning is returned.
- Neither isolation-clause nor lock-request-clause can be specified in the context where they will cause conflict isolation or lock intent on a common table expression (SQLSTATE 42601). This restriction does not apply to aliases or base tables. The following examples create an isolation conflict on $\sigma$ and returns an error:
  - **View:**
    ```sql
    create view a as (...);
    (select * from a with RR USE AND KEEP SHARE LOCKS)
    UNION ALL
    (select * from a with UR);
    ```
  - **Common table expression:**
    ```sql
    WITH a as (...)
    (select * from a with RR USE AND KEEP SHARE LOCKS)
    UNION ALL
    (select * from a with UR);
    ```
- An isolation-clause cannot be specified in an XML context (SQLSTATE 2200M).

**Examples of subselects**

**Example A1:** Select all columns and rows from the EMPLOYEE table.

```sql
SELECT * FROM EMPLOYEE
```

**Example A2:** Join the EMP_ACT and EMPLOYEE tables, select all the columns from the EMP_ACT table and add the employee’s surname (LASTNAME) from the EMPLOYEE table to each row of the result.

```sql
SELECT *
FROM EMPLOYEE
UNION ALL
(SELECT * FROM EMP_ACT)
UNION ALL
(SELECT LASTNAME FROM EMPLOYEE)
```
Example A3: Join the EMPLOYEE and DEPARTMENT tables, select the employee number (EMPNO), employee surname (LASTNAME), department number (WORKDEPT in the EMPLOYEE table and DEPTNO in the DEPARTMENT table) and department name (DEPTNAME) of all employees who were born (BIRTHDATE) earlier than 1930.
```
SELECT EMPNO, LASTNAME, WORKDEPT, DEPTNAME
FROM EMPLOYEE, DEPARTMENT
WHERE WORKDEPT = DEPTNO
AND YEAR(BIRTHDATE) < 1930
```

Example A4: Select the job (JOB) and the minimum and maximum salaries (SALARY) for each group of rows with the same job code in the EMPLOYEE table, but only for groups with more than one row and with a maximum salary greater than or equal to 27000.
```
SELECT JOB, MIN(SALARY), MAX(SALARY)
FROM EMPLOYEE
GROUP BY JOB
HAVING COUNT(*) > 1
AND MAX(SALARY) >= 27000
```

Example A5: Select all the rows of EMP.ACT table for employees (EMPNO) in department (WORKDEPT) 'E11'. (Employee department numbers are shown in the EMPLOYEE table.)
```
SELECT *
FROM EMP.ACT
WHERE EMPNO IN
(SELECT EMPNO
FROM EMPLOYEE
WHERE WORKDEPT = 'E11')
```

Example A6: From the EMPLOYEE table, select the department number (WORKDEPT) and maximum departmental salary (SALARY) for all departments whose maximum salary is less than the average salary for all employees.
```
SELECT WORKDEPT, MAX(SALARY)
FROM EMPLOYEE
GROUP BY WORKDEPT
HAVING MAX(SALARY) < (SELECT AVG(SALARY)
FROM EMPLOYEE)
```

The subquery in the HAVING clause would only be executed once in this example.

Example A7: Using the EMPLOYEE table, select the department number (WORKDEPT) and maximum departmental salary (SALARY) for all departments whose maximum salary is less than the average salary in all other departments.
```
SELECT WORKDEPT, MAX(SALARY)
FROM EMPLOYEE EMP_COR
GROUP BY WORKDEPT
HAVING MAX(SALARY) < (SELECT AVG(SALARY)
FROM EMPLOYEE
WHERE NOT WORKDEPT = EMP_COR.WORKDEPT)
```

In contrast to Example A6, the subquery in the HAVING clause would need to be executed for each group.
Example A8: Determine the employee number and salary of sales representatives along with the average salary and head count of their departments.

This query must first create a nested table expression (DINFO) in order to get the AVGSALARY and EMPCOUNT columns, as well as the DEPTNO column that is used in the WHERE clause.

```sql
SELECT THIS_EMP.EMPNO, THIS_EMP.SALARY, DINFO.AVGSALARY, DINFO.EMPCOUNT
FROM EMPLOYEE THIS_EMP,
     (SELECT OTHERS.WORKDEPT AS DEPTNO,
          AVG(OTHERS.SALARY) AS AVGSALARY,
          COUNT(*) AS EMPCOUNT
     FROM EMPLOYEE OTHERS
     GROUP BY OTHERS.WORKDEPT
     ) AS DINFO
WHERE THIS_EMP.JOB = 'SALESREP'
AND THIS_EMP.WORKDEPT = DINFO.DEPTNO
```

Using a nested table expression for this case saves the overhead of creating the DINFO view as a regular view. During statement preparation, accessing the catalog for the view is avoided and, because of the context of the rest of the query, only the rows for the department of the sales representatives need to be considered by the view.

Example A9: Display the average education level and salary for 5 random groups of employees.

This query requires the use of a nested table expression to set a random value for each employee so that it can subsequently be used in the GROUP BY clause.

```sql
SELECT RANDID, AVG(EDLEVEL), AVG(SALARY)
FROM ( SELECT EDLEVEL, SALARY, INTEGER(RAND()*5) AS RANDID
       FROM EMPLOYEE
       ) AS EMPRAND
GROUP BY RANDID
```

Example A10: Query the EMP_ACT table and return those project numbers that have an employee whose salary is in the top 10 of all employees.

```sql
SELECT EMP_ACT.EMPNO, PROJNO
FROM EMP_ACT
WHERE EMP_ACT.EMPNO IN
     (SELECT EMPLOYEE.EMPNO
      FROM EMPLOYEE
      ORDER BY SALARY DESC
      FETCH FIRST 10 ROWS ONLY)
```

Example A11: Assuming that PHONES and IDS are two SQL variables with array values of the same cardinality, turn these arrays into a table with three columns (one for each array and one for the position), and one row per array element.

```sql
SELECT T.PHONE, T.ID, T.INDEX FROM UNNEST(PHONES, IDS)
WITH ORDINALITY AS T(PHONE, ID, INDEX)
ORDER BY T.INDEX
```

Examples of joins

Example B1: This example illustrates the results of the various joins using tables J1 and J2. These tables contain rows as shown.

```sql
SELECT * FROM J1
W X
--- ------
```
The following query does an inner join of J1 and J2 matching the first column of both tables.

```
SELECT * FROM J1 INNER JOIN J2 ON W=Y
```

In this inner join example the row with column W='C' from J1 and the row with column Y='D' from J2 are not included in the result because they do not have a match in the other table. Note that the following alternative form of an inner join query produces the same result.

```
SELECT * FROM J1, J2 WHERE W=Y
```

The following left outer join will get back the missing row from J1 with nulls for the columns of J2. Every row from J1 is included.

```
SELECT * FROM J1 LEFT OUTER JOIN J2 ON W=Y
```

The following right outer join will get back the missing row from J2 with nulls for the columns of J1. Every row from J2 is included.

```
SELECT * FROM J1 RIGHT OUTER JOIN J2 ON W=Y
```

The following full outer join will get back the missing rows from both J1 and J2 with nulls where appropriate. Every row from both J1 and J2 is included.

```
SELECT * FROM J1 FULL OUTER JOIN J2 ON W=Y
```

Example B2: Using the tables J1 and J2 from the previous example, examine what happens when an additional predicate is added to the search condition.
The additional condition caused the inner join to select only 1 row compared to the inner join in Example B1.

Notice what the impact of this is on the full outer join.

<table>
<thead>
<tr>
<th>W</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>13</td>
<td>£</td>
<td>22</td>
</tr>
</tbody>
</table>

The result now has 5 rows (compared to 4 without the additional predicate) since there was only 1 row in the inner join and all rows of both tables must be returned.

The following query illustrates that placing the same additional predicate in WHERE clause has completely different results.

<table>
<thead>
<tr>
<th>W</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>A</td>
<td>21</td>
</tr>
<tr>
<td>C</td>
<td>13</td>
<td>£</td>
<td>22</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>D</td>
<td>23</td>
</tr>
<tr>
<td>A</td>
<td>11</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>B</td>
<td>12</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

The WHERE clause is applied after the intermediate result of the full outer join. This intermediate result would be the same as the result of the full outer join query in Example B1. The WHERE clause is applied to this intermediate result and eliminates all but the row that has X=13. Choosing the location of a predicate when performing outer joins can have significant impact on the results. Consider what happens if the predicate was X=12 instead of X=13. The following inner join returns no rows.

<table>
<thead>
<tr>
<th>W</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>13</td>
<td>£</td>
<td>22</td>
</tr>
</tbody>
</table>

Hence, the full outer join would return 6 rows, 3 from J1 with nulls for the columns of J2 and 3 from J2 with nulls for the columns of J1.

If the additional predicate is in the WHERE clause instead, 1 row is returned.
Example B3: List every department with the employee number and last name of the manager, including departments without a manager.

```
SELECT DEPTNO, DEPTNAME, EMPNO, LASTNAME
FROM DEPARTMENT LEFT OUTER JOIN EMPLOYEE
ON MGRNO = EMPNO
```

Example B4: List every employee number and last name with the employee number and last name of their manager, including employees without a manager.

```
SELECT E.EMPNO, E.LASTNAME, M.EMPNO, M.LASTNAME
FROM EMPLOYEE E LEFT OUTER JOIN
DEPARTMENT INNER JOIN EMPLOYEE M
ON MGRNO = M.EMPNO
ON E.WORKDEPT = DEPTNO
```

The inner join determines the last name for any manager identified in the DEPARTMENT table and the left outer join guarantees that each employee is listed even if a corresponding department is not found in DEPARTMENT.

**Examples of grouping sets, cube, and rollup**

The queries in Example C1 through Example C4 use a subset of the rows in the SALES tables based on the predicate 'WEEK(SALES_DATE) = 13'.

```
SELECT WEEK(SALES_DATE) AS WEEK,
       DAYOFWEEK(SALES_DATE) AS DAY_WEEK,
       SALES_PERSON, SALES AS UNITS_SOLD
FROM SALES
WHERE WEEK(SALES_DATE) = 13
```

which results in:

<table>
<thead>
<tr>
<th>WEEK</th>
<th>DAY_WEEK</th>
<th>SALES_PERSON</th>
<th>UNITS_SOLD</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>6</td>
<td>LUCHESSI</td>
<td>3</td>
</tr>
<tr>
<td>13</td>
<td>6</td>
<td>LUCHESSI</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>6</td>
<td>LEE</td>
<td>2</td>
</tr>
<tr>
<td>13</td>
<td>6</td>
<td>LEE</td>
<td>2</td>
</tr>
<tr>
<td>13</td>
<td>6</td>
<td>LEE</td>
<td>3</td>
</tr>
<tr>
<td>13</td>
<td>6</td>
<td>LEE</td>
<td>5</td>
</tr>
<tr>
<td>13</td>
<td>6</td>
<td>GOUNOT</td>
<td>3</td>
</tr>
<tr>
<td>13</td>
<td>6</td>
<td>GOUNOT</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>6</td>
<td>GOUNOT</td>
<td>7</td>
</tr>
<tr>
<td>13</td>
<td>7</td>
<td>LUCHESSI</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>7</td>
<td>LUCHESSI</td>
<td>2</td>
</tr>
<tr>
<td>13</td>
<td>7</td>
<td>LUCHESSI</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>7</td>
<td>LEE</td>
<td>7</td>
</tr>
<tr>
<td>13</td>
<td>7</td>
<td>LEE</td>
<td>3</td>
</tr>
<tr>
<td>13</td>
<td>7</td>
<td>LEE</td>
<td>7</td>
</tr>
<tr>
<td>13</td>
<td>7</td>
<td>LEE</td>
<td>4</td>
</tr>
<tr>
<td>13</td>
<td>7</td>
<td>GOUNOT</td>
<td>2</td>
</tr>
<tr>
<td>13</td>
<td>7</td>
<td>GOUNOT</td>
<td>18</td>
</tr>
<tr>
<td>13</td>
<td>7</td>
<td>GOUNOT</td>
<td>1</td>
</tr>
</tbody>
</table>

Example C1: Here is a query with a basic GROUP BY clause over 3 columns:
SELECT WEEK(SALES_DATE) AS WEEK,
       DAYOFWEEK(SALES_DATE) AS DAY_WEEK,
       SALES_PERSON, SUM(SALES) AS UNITS_SOLD
FROM SALES
WHERE WEEK(SALES_DATE) = 13
GROUP BY WEEK(SALES_DATE), DAYOFWEEK(SALES_DATE), SALES_PERSON
ORDER BY WEEK, DAY_WEEK, SALES_PERSON

This results in:

<table>
<thead>
<tr>
<th>WEEK</th>
<th>DAY_WEEK</th>
<th>SALES_PERSON</th>
<th>UNITS_SOLD</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>6</td>
<td>GOUNOT</td>
<td>11</td>
</tr>
<tr>
<td>13</td>
<td>6</td>
<td>LEE</td>
<td>12</td>
</tr>
<tr>
<td>13</td>
<td>6</td>
<td>LUCCHESSI</td>
<td>4</td>
</tr>
<tr>
<td>13</td>
<td>7</td>
<td>GOUNOT</td>
<td>21</td>
</tr>
<tr>
<td>13</td>
<td>7</td>
<td>LEE</td>
<td>21</td>
</tr>
<tr>
<td>13</td>
<td>7</td>
<td>LUCCHESSI</td>
<td>4</td>
</tr>
</tbody>
</table>

Example C2: Produce the result based on two different grouping sets of rows from the SALES table.

SELECT WEEK(SALES_DATE) AS WEEK,
       DAYOFWEEK(SALES_DATE) AS DAY_WEEK,
       SALES_PERSON, SUM(SALES) AS UNITS_SOLD
FROM SALES
WHERE WEEK(SALES_DATE) = 13
GROUP BY GROUPING SETS
        ( (WEEK(SALES_DATE), SALES_PERSON),
          (DAYOFWEEK(SALES_DATE), SALES_PERSON))
ORDER BY WEEK, DAY_WEEK, SALES_PERSON

This results in:

<table>
<thead>
<tr>
<th>WEEK</th>
<th>DAY_WEEK</th>
<th>SALES_PERSON</th>
<th>UNITS_SOLD</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>-</td>
<td>GOUNOT</td>
<td>32</td>
</tr>
<tr>
<td>13</td>
<td>-</td>
<td>LEE</td>
<td>33</td>
</tr>
<tr>
<td>13</td>
<td>-</td>
<td>LUCCHESSI</td>
<td>8</td>
</tr>
<tr>
<td>-</td>
<td>6</td>
<td>GOUNOT</td>
<td>11</td>
</tr>
<tr>
<td>-</td>
<td>6</td>
<td>LEE</td>
<td>12</td>
</tr>
<tr>
<td>-</td>
<td>6</td>
<td>LUCCHESSI</td>
<td>4</td>
</tr>
<tr>
<td>-</td>
<td>7</td>
<td>GOUNOT</td>
<td>21</td>
</tr>
<tr>
<td>-</td>
<td>7</td>
<td>LEE</td>
<td>21</td>
</tr>
<tr>
<td>-</td>
<td>7</td>
<td>LUCCHESSI</td>
<td>4</td>
</tr>
</tbody>
</table>

The rows with WEEK 13 are from the first grouping set and the other rows are from the second grouping set.

Example C3: If you use the 3 distinct columns involved in the grouping sets of Example C2 and perform a ROLLUP, you can see grouping sets for (WEEK,DAY_WEEK,SALES_PERSON), (WEEK, DAY_WEEK), (WEEK) and grand total.

SELECT WEEK(SALES_DATE) AS WEEK,
       DAYOFWEEK(SALES_DATE) AS DAY_WEEK,
       SALES_PERSON, SUM(SALES) AS UNITS_SOLD
FROM SALES
WHERE WEEK(SALES_DATE) = 13
GROUP BY ROLLUP
        ( WEEK(SALES_DATE), DAYOFWEEK(SALES_DATE), SALES_PERSON )
ORDER BY WEEK, DAY_WEEK, SALES_PERSON

This results in:

<table>
<thead>
<tr>
<th>WEEK</th>
<th>DAY_WEEK</th>
<th>SALES_PERSON</th>
<th>UNITS_SOLD</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>6</td>
<td>GOUNOT</td>
<td>11</td>
</tr>
<tr>
<td>13</td>
<td>6</td>
<td>LEE</td>
<td>12</td>
</tr>
</tbody>
</table>

subselect
Example C4: If you run the same query as Example C3 only replace ROLLUP with CUBE, you can see additional grouping sets for (WEEK,SALES_PERSON), (DAY_WEEK,SALES_PERSON), (DAY_WEEK), (SALES_PERSON) in the result.

```
SELECT WEEK(SALES_DATE) AS WEEK, DAYOFWEEK(SALES_DATE) AS DAY_WEEK, SALES_PERSON, SUM(SALES) AS UNITS_SOLD
FROM SALES
WHERE WEEK(SALES_DATE) = 13
GROUP BY CUBE (WEEK(SALES_DATE), DAYOFWEEK(SALES_DATE), SALES_PERSON)
ORDER BY WEEK, DAY_WEEK, SALES_PERSON
```

This results in:

<table>
<thead>
<tr>
<th>WEEK</th>
<th>DAY_WEEK</th>
<th>SALES_PERSON</th>
<th>UNITS_SOLD</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>6</td>
<td>GOUNOT</td>
<td>11</td>
</tr>
<tr>
<td>13</td>
<td>6</td>
<td>LEE</td>
<td>12</td>
</tr>
<tr>
<td>13</td>
<td>6</td>
<td>LUCHESSI</td>
<td>4</td>
</tr>
<tr>
<td>13</td>
<td>6</td>
<td></td>
<td>27</td>
</tr>
<tr>
<td>13</td>
<td>7</td>
<td>GOUNOT</td>
<td>21</td>
</tr>
<tr>
<td>13</td>
<td>7</td>
<td>LEE</td>
<td>21</td>
</tr>
<tr>
<td>13</td>
<td>7</td>
<td>LUCHESSI</td>
<td>4</td>
</tr>
<tr>
<td>13</td>
<td>7</td>
<td></td>
<td>46</td>
</tr>
<tr>
<td>13</td>
<td>-</td>
<td>GOUNOT</td>
<td>32</td>
</tr>
<tr>
<td>13</td>
<td>-</td>
<td>LEE</td>
<td>33</td>
</tr>
<tr>
<td>13</td>
<td>-</td>
<td>LUCHESSI</td>
<td>8</td>
</tr>
<tr>
<td>13</td>
<td>-</td>
<td></td>
<td>73</td>
</tr>
<tr>
<td>-</td>
<td>6</td>
<td>GOUNOT</td>
<td>11</td>
</tr>
<tr>
<td>-</td>
<td>6</td>
<td>LEE</td>
<td>12</td>
</tr>
<tr>
<td>-</td>
<td>6</td>
<td>LUCHESSI</td>
<td>4</td>
</tr>
<tr>
<td>-</td>
<td>6</td>
<td></td>
<td>27</td>
</tr>
<tr>
<td>-</td>
<td>7</td>
<td>GOUNOT</td>
<td>21</td>
</tr>
<tr>
<td>-</td>
<td>7</td>
<td>LEE</td>
<td>21</td>
</tr>
<tr>
<td>-</td>
<td>7</td>
<td>LUCHESSI</td>
<td>4</td>
</tr>
<tr>
<td>-</td>
<td>7</td>
<td></td>
<td>46</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>GOUNOT</td>
<td>32</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>LEE</td>
<td>33</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>LUCHESSI</td>
<td>8</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td></td>
<td>73</td>
</tr>
</tbody>
</table>

Example C5: Obtain a result set which includes a grand-total of selected rows from the SALES table together with a group of rows aggregated by SALES_PERSON and MONTH.

```
SELECT SALES_PERSON, MONTH(SALES_DATE) AS MONTH, SUM(SALES) AS UNITS_SOLD
FROM SALES
GROUP BY GROUPING SETS ( (SALES_PERSON, MONTH(SALES_DATE)), ()
ORDER BY SALES_PERSON, MONTH
```

This results in:
Example C6: This example shows two simple ROLLUP queries followed by a query which treats the two ROLLUPs as grouping sets in a single result set and specifies row ordering for each column involved in the grouping sets.

Example C6-1:
```
SELECT WEEK(SALES_DATE) AS WEEK,
       DAYOFWEEK(SALES_DATE) AS DAY_WEEK,
       SUM(SALES) AS UNITS_SOLD
FROM SALES
GROUP BY ROLLUP
    ( WEEK(SALES_DATE), DAYOFWEEK(SALES_DATE) )
ORDER BY WEEK, DAY_WEEK
```
results in:
```
WEEK DAY_WEEK UNITS_SOLD
---------- ----------- -----------
13 6 27
13 7 46
13 - 73
14 1 31
14 2 43
14 - 74
53 1 8
53 - 8
- - 155
```

Example C6-2:
```
SELECT MONTH(SALES_DATE) AS MONTH,
       REGION,
       SUM(SALES) AS UNITS_SOLD
FROM SALES
GROUP BY ROLLUP
    ( MONTH(SALES_DATE), REGION )
ORDER BY MONTH, REGION
```
results in:
```
MONTH REGION UNITS_SOLD
---------- --------------- -----------
3 Manitoba 22
3 Ontario-North 8
3 Ontario-South 34
3 Quebec 40
3 - 104
4 Manitoba 17
4 Ontario-North 1
4 Ontario-South 14
4 Quebec 11
4 - 43
12 Manitoba 2
```
Example C6-3:

```sql
SELECT WEEK(SALES_DATE) AS WEEK,
       DAYOFWEEK(SALES_DATE) AS DAY_WEEK,
       MONTH(SALES_DATE) AS MONTH,
       REGION,
       SUM(SALES) AS UNITS_SOLD
FROM SALES
GROUP BY GROUPING SETS ( ROLLUP (WEEK(SALES_DATE), DAYOFWEEK(SALES_DATE)),
                          ROLLUP (MONTH(SALES_DATE), REGION))
ORDER BY WEEK, DAY_WEEK, MONTH, REGION
```

results in:

<table>
<thead>
<tr>
<th>WEEK</th>
<th>DAY_WEEK</th>
<th>MONTH</th>
<th>REGION</th>
<th>UNITS_SOLD</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>6</td>
<td>-</td>
<td>-</td>
<td>27</td>
</tr>
<tr>
<td>13</td>
<td>7</td>
<td>-</td>
<td>-</td>
<td>46</td>
</tr>
<tr>
<td>13</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>73</td>
</tr>
<tr>
<td>14</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>31</td>
</tr>
<tr>
<td>14</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>43</td>
</tr>
<tr>
<td>14</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>74</td>
</tr>
<tr>
<td>53</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>8</td>
</tr>
<tr>
<td>53</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>8</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>3</td>
<td>Manitoba</td>
<td>22</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>3</td>
<td>Ontario-North</td>
<td>8</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>3</td>
<td>Ontario-South</td>
<td>34</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>3</td>
<td>Quebec</td>
<td>40</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>3</td>
<td>-</td>
<td>104</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>4</td>
<td>Manitoba</td>
<td>17</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>4</td>
<td>Ontario-North</td>
<td>1</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>4</td>
<td>Ontario-South</td>
<td>14</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>4</td>
<td>Quebec</td>
<td>11</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>4</td>
<td>-</td>
<td>43</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>12</td>
<td>Manitoba</td>
<td>2</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>12</td>
<td>Ontario-South</td>
<td>4</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>12</td>
<td>Quebec</td>
<td>2</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>8</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>155</td>
</tr>
</tbody>
</table>

Using the two ROLLUPs as grouping sets causes the result to include duplicate rows. There are even two grand total rows.

Observe how the use of ORDER BY has affected the results:
- In the first grouped set, week 53 has been repositioned to the end.
- In the second grouped set, month 12 has now been positioned to the end and the regions now appear in alphabetic order.
- Null values are sorted high.

Example C7: In queries that perform multiple ROLLUPs in a single pass (such as Example C6-3) you may want to be able to indicate which grouping set produced each row. The following steps demonstrate how to provide a column (called GROUP) which indicates the origin of each row in the result set. By origin, we mean which one of the two grouping sets produced the row in the result set.
*Step 1:* Introduce a way of "generating" new data values, using a query which selects from a VALUES clause (which is an alternate form of a fullselect). This query shows how a table can be derived called "X" having 2 columns "R1" and "R2" and 1 row of data.

```sql
SELECT R1, R2
FROM (VALUES ('GROUP 1', 'GROUP 2')) AS X(R1, R2);
```

results in:

<table>
<thead>
<tr>
<th>R1</th>
<th>R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>GROUP 1</td>
<td>GROUP 2</td>
</tr>
</tbody>
</table>

*Step 2:* Form the cross product of this table "X" with the SALES table. This add columns "R1" and "R2" to every row.

```sql
SELECT R1, R2, WEEK(SALES_DATE) AS WEEK,
       DAYOFWEEK(SALES_DATE) AS DAY_WEEK,
       MONTH(SALES_DATE) AS MONTH,
       REGION,
       SUM(SALES) AS UNITS_SOLD
FROM SALES, (VALUES ('GROUP 1', 'GROUP 2')) AS X(R1, R2)
GROUP BY GROUPING SETS
         ((R1, R2, WEEK(SALES_DATE) AS WEEK, DAYOFWEEK(SALES_DATE) AS DAY_WEEK, MONTH(SALES_DATE) AS MONTH, REGION, SUM(SALES) AS UNITS_SOLD)
          FROM SALES, (VALUES ('GROUP 1', 'GROUP 2')) AS X(R1, R2)
         )
ORDER BY WEEK, DAY_WEEK, MONTH, REGION
```

Step 3: Now we can combine these columns with the grouping sets to include these columns in the rollup analysis.

```sql
SELECT R1, R2,
       WEEK(SALES_DATE) AS WEEK,
       DAYOFWEEK(SALES_DATE) AS DAY_WEEK,
       MONTH(SALES_DATE) AS MONTH,
       REGION, SUM(SALES) AS UNITS_SOLD
FROM SALES, (VALUES ('GROUP 1', 'GROUP 2')) AS X(R1, R2)
GROUP BY GROUPING SETS
         ((R1, ROLLUP(WEEK(SALES_DATE), DAYOFWEEK(SALES_DATE)),
           (R2, ROLLUP(MONTH(SALES_DATE), REGION))
         )
ORDER BY WEEK, DAY_WEEK, MONTH, REGION
```

results in:

<table>
<thead>
<tr>
<th>R1</th>
<th>R2</th>
<th>WEEK</th>
<th>DAY_WEEK</th>
<th>MONTH</th>
<th>REGION</th>
<th>UNITS_SOLD</th>
</tr>
</thead>
<tbody>
<tr>
<td>GROUP 1</td>
<td>-</td>
<td>13</td>
<td>6</td>
<td>-</td>
<td>-</td>
<td>27</td>
</tr>
<tr>
<td>GROUP 1</td>
<td>-</td>
<td>13</td>
<td>7</td>
<td>-</td>
<td>-</td>
<td>46</td>
</tr>
<tr>
<td>GROUP 1</td>
<td>-</td>
<td>13</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>73</td>
</tr>
<tr>
<td>GROUP 1</td>
<td>-</td>
<td>14</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>31</td>
</tr>
<tr>
<td>GROUP 1</td>
<td>-</td>
<td>14</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>43</td>
</tr>
<tr>
<td>GROUP 1</td>
<td>-</td>
<td>14</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>74</td>
</tr>
<tr>
<td>GROUP 1</td>
<td>-</td>
<td>53</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>8</td>
</tr>
<tr>
<td>GROUP 1</td>
<td>-</td>
<td>53</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>8</td>
</tr>
<tr>
<td>GROUP 2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>3</td>
<td>Manitoba</td>
<td>22</td>
</tr>
<tr>
<td>GROUP 2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>3</td>
<td>Ontario-North</td>
<td>8</td>
</tr>
<tr>
<td>GROUP 2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>3</td>
<td>Ontario-South</td>
<td>34</td>
</tr>
<tr>
<td>GROUP 2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>3</td>
<td>Quebec</td>
<td>40</td>
</tr>
<tr>
<td>GROUP 2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>3</td>
<td>-</td>
<td>104</td>
</tr>
<tr>
<td>GROUP 2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>4</td>
<td>Manitoba</td>
<td>17</td>
</tr>
<tr>
<td>GROUP 2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>4</td>
<td>Ontario-North</td>
<td>1</td>
</tr>
<tr>
<td>GROUP 2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>4</td>
<td>Ontario-South</td>
<td>14</td>
</tr>
<tr>
<td>GROUP 2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>4</td>
<td>Quebec</td>
<td>11</td>
</tr>
<tr>
<td>GROUP 2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>4</td>
<td>-</td>
<td>43</td>
</tr>
<tr>
<td>GROUP 2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>12</td>
<td>Manitoba</td>
<td>2</td>
</tr>
<tr>
<td>GROUP 2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>12</td>
<td>Ontario-South</td>
<td>4</td>
</tr>
<tr>
<td>GROUP 2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>12</td>
<td>Quebec</td>
<td>2</td>
</tr>
</tbody>
</table>
Step 4: Notice that because R1 and R2 are used in different grouping sets, whenever R1 is non-null in the result, R2 is null and whenever R2 is non-null in the result, R1 is null. That means you can consolidate these columns into a single column using the COALESCE function. You can also use this column in the ORDER BY clause to keep the results of the two grouping sets together.

```
SELECT COALESCE(R1, R2) AS GROUP,
       WEEK(SALES_DATE) AS WEEK,
       DAYOFWEEK(SALES_DATE) AS DAY_WEEK,
       MONTH(SALES_DATE) AS MONTH,
       REGION,
       SUM(SALES) AS UNITS_SOLD
FROM SALES, (VALUES ('GROUP 1', 'GROUP 2')) AS X(R1, R2)
GROUP BY GROUPING SETS ((R1, ROLLUP(WEEK(SALES_DATE),
                                        DAYOFWEEK(SALES_DATE)),
                           (R2, ROLLUP(MONTH(SALES_DATE), REGION)) )
ORDER BY GROUP, WEEK, DAY_WEEK, MONTH, REGION;
```

This results in:

```
GROUP  | WEEK | DAY_WEEK | MONTH | REGION | UNITS_SOLD
-------|------|----------|-------|--------|-------------
GROUP 1 | 13   | 6        | -     | -      | 27          
GROUP 1 | 13   | 7        | -     | -      | 46          
GROUP 1 | 13   | -        | -     | -      | 73          
GROUP 1 | 14   | 1        | -     | -      | 31          
GROUP 1 | 14   | 2        | -     | -      | 43          
GROUP 1 | 14   | -        | -     | -      | 74          
GROUP 1 | 53   | 1        | -     | -      | 8           
GROUP 1 | 53   | -        | -     | -      | 8           
GROUP 1 | -    | -        | -     | -      | 155         
GROUP 2 | -    | -        | 3     | Manitoba | 22         
GROUP 2 | -    | -        | 3     | Ontario-North | 8        
GROUP 2 | -    | -        | 3     | Ontario-South | 34       
GROUP 2 | -    | -        | 3     | Quebec     | 40         
GROUP 2 | -    | -        | 3     | -         | 104        
GROUP 2 | -    | -        | 4     | Manitoba   | 17         
GROUP 2 | -    | -        | 4     | Ontario-North | 1        
GROUP 2 | -    | -        | 4     | Ontario-South | 14       
GROUP 2 | -    | -        | 4     | Quebec     | 11         
GROUP 2 | -    | -        | 4     | -          | 43         
GROUP 2 | -    | -        | 12    | Manitoba   | 2          
GROUP 2 | -    | -        | 12    | Ontario-South | 4        
GROUP 2 | -    | -        | 12    | Quebec     | 2          
GROUP 2 | -    | -        | 12    | -          | 8          
GROUP 2 | -    | -        | -     | -          | 155        
```

Example C8: The following example illustrates the use of various aggregate functions when performing a CUBE. The example also makes use of cast functions and rounding to produce a decimal result with reasonable precision and scale.

```
SELECT MONTH(SALES_DATE) AS MONTH,
       REGION,
       SUM(SALES) AS UNITS_SOLD,
       MAX(SALES) AS BEST_SALE,
       CAST(ROUND(AVG(DECIMAL(SALES)), 2) AS DECIMAL(5, 2)) AS AVG_UNITS_SOLD
FROM SALES
GROUP BY CUBE(MONTH(SALES_DATE), REGION)
ORDER BY MONTH, REGION
```

This results in:
<table>
<thead>
<tr>
<th>MONTH</th>
<th>REGION</th>
<th>UNITS_SOLD</th>
<th>BEST_SALE</th>
<th>AVG_UNITS_SOLD</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Manitoba</td>
<td>22</td>
<td>7</td>
<td>3.14</td>
</tr>
<tr>
<td>3</td>
<td>Ontario-North</td>
<td>8</td>
<td>3</td>
<td>2.67</td>
</tr>
<tr>
<td>3</td>
<td>Ontario-South</td>
<td>34</td>
<td>14</td>
<td>4.25</td>
</tr>
<tr>
<td>3</td>
<td>Quebec</td>
<td>40</td>
<td>18</td>
<td>5.00</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>104</td>
<td>18</td>
<td>4.00</td>
</tr>
<tr>
<td>4</td>
<td>Manitoba</td>
<td>17</td>
<td>9</td>
<td>5.67</td>
</tr>
<tr>
<td>4</td>
<td>Ontario-North</td>
<td>1</td>
<td>1</td>
<td>1.00</td>
</tr>
<tr>
<td>4</td>
<td>Ontario-South</td>
<td>14</td>
<td>8</td>
<td>4.67</td>
</tr>
<tr>
<td>4</td>
<td>Quebec</td>
<td>11</td>
<td>8</td>
<td>5.50</td>
</tr>
<tr>
<td>4</td>
<td>-</td>
<td>43</td>
<td>9</td>
<td>4.78</td>
</tr>
<tr>
<td>12</td>
<td>Manitoba</td>
<td>2</td>
<td>2</td>
<td>2.00</td>
</tr>
<tr>
<td>12</td>
<td>Ontario-South</td>
<td>4</td>
<td>3</td>
<td>2.00</td>
</tr>
<tr>
<td>12</td>
<td>Quebec</td>
<td>2</td>
<td>1</td>
<td>1.00</td>
</tr>
<tr>
<td>12</td>
<td>-</td>
<td>8</td>
<td>3</td>
<td>1.60</td>
</tr>
<tr>
<td>-</td>
<td>Manitoba</td>
<td>41</td>
<td>9</td>
<td>3.73</td>
</tr>
<tr>
<td>-</td>
<td>Ontario-North</td>
<td>9</td>
<td>3</td>
<td>2.25</td>
</tr>
<tr>
<td>-</td>
<td>Ontario-South</td>
<td>52</td>
<td>14</td>
<td>4.00</td>
</tr>
<tr>
<td>-</td>
<td>Quebec</td>
<td>53</td>
<td>18</td>
<td>4.42</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>155</td>
<td>18</td>
<td>3.87</td>
</tr>
</tbody>
</table>
The fullselect is a component of the select-statement, the INSERT statement, and the CREATE VIEW statement. It is also a component of certain predicates which, in turn, are components of a statement. A fullselect that is a component of a predicate is called a subquery, and a fullselect that is enclosed in parentheses is sometimes called a subquery.

The set operators UNION, EXCEPT, and INTERSECT correspond to the relational operators union, difference, and intersection.

A fullselect specifies a result table. If a set operator is not used, the result of the fullselect is the result of the specified subselect or values-clause.

The authorization for a fullselect is described in the Authorization section in "SQL queries".

values-clause

Derives a result table by specifying the actual values, using expressions or row expressions, for each column of a row in the result table. Multiple rows may be specified. The result type of any expression in the values-clause cannot be a row type (SQLSTATE 428H2).
NULL can only be used with multiple specifications of values-row, either as the column value of a single column result table or within a row-expression, and at least one row in the same column must not be NULL (SQLSTATE 42608).

A values-row is specified by:
- A single expression for a single column result table
- \(n\) expressions (or NULL) separated by commas and enclosed in parentheses, where \(n\) is the number of columns in the result table or, a row expression for a multiple column result table.

A multiple row VALUES clause must have the same number of columns in each values-row (SQLSTATE 42826).

The following are examples of values-clause and their meaning.
- VALUES (1),(2),(3) - 3 rows of 1 column
- VALUES 1, 2, 3 - 3 rows of 1 column
- VALUES (1, 2, 3) - 1 row of 3 columns
- VALUES (1,21),(2,22),(3,23) - 3 rows of 2 columns

A values-clause that is composed of \(n\) specifications of values-row, \(R_{E_1} to R_{E_n}\), where \(n\) is greater than 1, is equivalent to:

\[RE_1 \text{ UNION ALL } RE_2 \ldots \text{ UNION ALL } RE_n\]

This means that the corresponding columns of each values-row must be comparable (SQLSTATE 42825).

**UNION** or **UNION ALL**

Derives a result table by combining two other result tables (R1 and R2). If UNION ALL is specified, the result consists of all rows in R1 and R2. If UNION is specified without the ALL option, the result is the set of all rows in either R1 or R2, with the duplicate rows eliminated. In either case, however, each row of the UNION table is either a row from R1 or a row from R2.

**EXCEPT** or **EXCEPT ALL**

Derives a result table by combining two other result tables (R1 and R2). If EXCEPT ALL is specified, the result consists of all rows that do not have a corresponding row in R2, where duplicate rows are significant. If EXCEPT is specified without the ALL option, the result consists of all rows that are only in R1, with duplicate rows in the result of this operation eliminated.

For compatibility with other SQL implementations, MINUS can be specified as a synonym for EXCEPT.

**INTERSECT** or **INTERSECT ALL**

Derives a result table by combining two other result tables (R1 and R2). If INTERSECT ALL is specified, the result consists of all rows that are in both R1 and R2. If INTERSECT is specified without the ALL option, the result consists of all rows that are in both R1 and R2, with the duplicate rows eliminated.

**order-by-clause**

See “subselect” for details of the order-by-clause. A fullselect that contains an ORDER BY clause cannot be specified in (SQLSTATE 428FJ):
- A materialized query table
- The outermost fullselect of a view

**Note:** An ORDER BY clause in a fullselect does not affect the order of the rows returned by a query. An ORDER BY clause only affects the order of the rows returned if it is specified in the outermost fullselect.
**fullselect**

**fetch-first-clause**

See “subselect” for details of the *fetch-first-clause*. A fullselect that contains a FETCH FIRST clause cannot be specified in (SQLSTATE 428FJ):

- A materialized query table
- The outermost fullselect of a view

**Note:** A FETCH FIRST clause in a fullselect does not affect the number of rows returned by a query. A FETCH FIRST clause only affects the number of rows returned if it is specified in the outermost fullselect.

**isolation-clause**

See “subselect” for details of the *isolation-clause*. If *isolation-clause* is specified for a fullselect and it could apply equally to a subselect of the fullselect, *isolation-clause* is applied to the fullselect. For example, consider the following query.

```sql
SELECT NAME FROM PRODUCT
UNION
SELECT NAME FROM CATALOG WITH UR
```

Even though the isolation clause WITH UR could apply only to the subselect SELECT NAME FROM CATALOG, it is applied to the whole fullselect.

The number of columns in the result tables R1 and R2 must be the same (SQLSTATE 42826). If the ALL keyword is not specified, R1 and R2 must not include any columns having a data type of CLOB, DBCLOB, BLOB, distinct type on any of these types, or structured type (SQLSTATE 42907).

The columns of the result are named as follows:

- If the *n*th column of R1 and the *n*th column of R2 have the same result column name, then the *n*th column of R has the result column name.
- If the *n*th column of R1 and the *n*th column of R2 have different result column names, a name is generated. This name cannot be used as the column name in an ORDER BY or UPDATE clause.

The generated name can be determined by performing a DESCRIBE of the SQL statement and consulting the SQLNAME field.

**Duplicate rows:** Two rows are duplicates if each value in the first is equal to the corresponding value of the second. For determining duplicates, two null values are considered equal, and two decimal floating-point representations of the same number are considered equal. For example, 2.00 and 2.0 have the same value (2.00 and 2.0 compare as equal) but have different exponents, which allows you to represent both 2.00 and 2.0. So, for example, if the result table of a UNION operation contains a decimal floating-point column and multiple representations of the same number exist, the one that is returned (for example, 2.00 or 2.0) is unpredictable. For more information, see “Numeric comparisons” on page 123.

When multiple operations are combined in an expression, operations within parentheses are performed first. If there are no parentheses, the operations are performed from left to right with the exception that all INTERSECT operations are performed before UNION or EXCEPT operations.

In the following example, the values of tables R1 and R2 are shown on the left. The other headings listed show the values as a result of various set operations on R1 and R2.
**Examples of a fullselect**

*Example 1:* Select all columns and rows from the EMPLOYEE table.

```sql
SELECT * FROM EMPLOYEE
```

*Example 2:* List the employee numbers (EMPNO) of all employees in the EMPLOYEE table whose department number (WORKDEPT) either begins with 'E' or who are assigned to projects in the EMP_ACT table whose project number (PROJNO) equals 'MA2100', 'MA2110', or 'MA2112'.

```sql
SELECT EMPNO FROM EMPLOYEE WHERE WORKDEPT LIKE 'E'
UNION
SELECT EMPNO FROM EMP_ACT WHERE PROJNO IN ('MA2100', 'MA2110', 'MA2112')
```

*Example 3:* Make the same query as in example 2, and, in addition, “tag” the rows from the EMPLOYEE table with 'emp' and the rows from the EMP_ACT table with 'emp_act'. Unlike the result from example 2, this query may return the same EMPNO more than once, identifying which table it came from by the associated “tag”.

```sql
SELECT EMPNO, 'emp' FROM EMPLOYEE WHERE WORKDEPT LIKE 'E'
UNION
SELECT EMPNO, 'emp_act' FROM EMP_ACT WHERE PROJNO IN ('MA2100', 'MA2110', 'MA2112')
```
Example 4: Make the same query as in example 2, only use UNION ALL so that no duplicate rows are eliminated.

```sql
SELECT EMPNO
FROM EMPLOYEE
    WHERE WORKDEPT LIKE 'E%'
UNION ALL
SELECT EMPNO
FROM EMP_ACT
    WHERE PROJNO IN ('MA2100', 'MA2110', 'MA2112')
```

Example 5: Make the same query as in Example 3, only include an additional two employees currently not in any table and tag these rows as "new".

```sql
SELECT EMPNO, 'emp'
FROM EMPLOYEE
    WHERE WORKDEPT LIKE 'E%
UNION
SELECT EMPNO, 'emp_act'
FROM EMP_ACT
    WHERE PROJNO IN ('MA2100', 'MA2110', 'MA2112')
UNION
VALUES ('NEWAAA', 'new'), ('NEWBBB', 'new')
```

Example 6: This example of EXCEPT produces all rows that are in T1 but not in T2.

```sql
(SELECT * FROM T1)
EXCEPT ALL
(SELECT * FROM T2)
```

If no NULL values are involved, this example returns the same results as

```sql
SELECT ALL *
FROM T1
WHERE NOT EXISTS (SELECT * FROM T2
    WHERE T1.C1 = T2.C1 AND T1.C2 = T2.C2 AND...)
```

Example 7: This example of INTERSECT produces all rows that are in both tables T1 and T2, removing duplicates.

```sql
(SELECT * FROM T1)
INTERSECT
(SELECT * FROM T2)
```

If no NULL values are involved, this example returns the same result as

```sql
SELECT DISTINCT *
FROM T1
WHERE EXISTS (SELECT * FROM T2
    WHERE T1.C1 = T2.C1 AND T1.C2 = T2.C2 AND...)
```

where C1, C2, and so on represent the columns of T1 and T2.
The **select-statement** is the form of a query that can be directly specified in a DECLARE CURSOR statement, or prepared and then referenced in a DECLARE CURSOR statement. It can also be issued through the use of dynamic SQL statements using the command line processor (or similar tools), causing a result table to be displayed on the user’s screen. In either case, the table specified by a select-statement is the result of the fullselect.

The authorization for a select-statement is described in the Authorization section in “SQL queries”.

### common-table-expression

The **common-table-expression** permits defining a result table with a table-name that can be specified as a table name in any FROM clause of the fullselect that follows. Multiple common table expressions can be specified following the single WITH keyword. Each common table expression specified can also be referenced by name in the FROM clause of subsequent common table expressions.

If a list of columns is specified, it must consist of as many names as there are columns in the result table of the fullselect. Each column-name must be unique and unqualified. If these column names are not specified, the names are derived from the select list of the fullselect used to define the common table expression.

The table-name of a common table expression must be different from any other common table expression table-name in the same statement (SQLSTATE 42726). If the common table expression is specified in an INSERT statement the table-name cannot be the same as the table or view name that is the object of the insert (SQLSTATE 42726). A common table expression table-name can be specified as a table name in any FROM clause throughout the fullselect. A table-name of a common table expression overrides any existing table, view or alias (in the catalog) with the same qualified name.

If more than one common table expression is defined in the same statement, cyclic references between the common table expressions are not permitted (SQLSTATE 42726).
A cyclic reference occurs when two common table expressions \( dt1 \) and \( dt2 \) are created such that \( dt1 \) refers to \( dt2 \) and \( dt2 \) refers to \( dt1 \).

If the fullselect of a common table expression contains a \textit{data-change-table-reference} in the FROM clause, the common table expression is said to modify data. A common table expression that modifies data is always evaluated when the statement is processed, regardless of whether the common table expression is used anywhere else in the statement. If there is at least one common table expression that reads or modifies data, all common table expressions are processed in the order in which they occur, and each common table expression that reads or modifies data is completely executed, including all constraints and triggers, before any subsequent common table expressions are executed.

The common table expression is also optional prior to the fullselect in the CREATE VIEW and INSERT statements.

A common table expression can be used:
- In place of a view to avoid creating the view (when general use of the view is not required and positioned updates or deletes are not used)
- To enable grouping by a column that is derived from a scalar subselect or function that is not deterministic or has external action
- When the desired result table is based on host variables
- When the same result table needs to be shared in a fullselect
- When the result needs to be derived using recursion
- When multiple SQL data change statements need to be processed within the query

If the fullselect of a common table expression contains a reference to itself in a FROM clause, the common table expression is a \textit{recursive common table expression}. Queries using recursion are useful in supporting applications such as bill of materials (BOM), reservation systems, and network planning.

The following must be true of a recursive common table expression:
- Each fullselect that is part of the recursion cycle must start with SELECT or SELECT ALL. Use of SELECT DISTINCT is not allowed (SQLSTATE 42925). Furthermore, the unions must use UNION ALL (SQLSTATE 42925).
- The column names must be specified following the \textit{table-name} of the common table expression (SQLSTATE 42908).
- The first fullselect of the first union (the initialization fullselect) must not include a reference to any column of the common table expression in any FROM clause (SQLSTATE 42836).
- If a column name of the common table expression is referred to in the iterative fullselect, the data type, length, and code page for the column are determined based on the initialization fullselect. The corresponding column in the iterative fullselect must have the same data type and length as the data type and length determined based on the initialization fullselect and the code page must match (SQLSTATE 42825). However, for character string types, the length of the two data types may differ. In this case, the column in the iterative fullselect must have a length that would always be assignable to the length determined from the initialization fullselect.
- Each fullselect that is part of the recursion cycle must not include any aggregate functions, group-by-clauses, or having-clauses (SQLSTATE 42836).
The FROM clauses of these fullselects can include at most one reference to a common table expression that is part of a recursion cycle (SQLSTATE 42836).

- The iterative fullselect and the overall recursive fullselect must not include an order-by-clause (SQLSTATE 42836).
- Subqueries (scalar or quantified) must not be part of any recursion cycles (SQLSTATE 42836).

When developing recursive common table expressions, remember that an infinite recursion cycle (loop) can be created. Check that recursion cycles will terminate. This is especially important if the data involved is cyclic. A recursive common table expression is expected to include a predicate that will prevent an infinite loop. The recursive common table expression is expected to include:

- In the iterative fullselect, an integer column incremented by a constant.
- A predicate in the where clause of the iterative fullselect in the form "counter_col < constant" or "counter_col < :hostvar".

A warning is issued if this syntax is not found in the recursive common table expression (SQLSTATE 01605).

**Recursion example: bill of materials**

Bill of materials (BOM) applications are a common requirement in many business environments. To illustrate the capability of a recursive common table expression for BOM applications, consider a table of parts with associated subparts and the quantity of subparts required by the part. For this example, create the table as follows:

```sql
CREATE TABLE PARTLIST
(PART VARCHAR(8),
SUBPART VARCHAR(8),
QUANTITY INTEGER);
```

To give query results for this example, assume that the PARTLIST table is populated with the following values:

<table>
<thead>
<tr>
<th>PART</th>
<th>SUBPART</th>
<th>QUANTITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>01</td>
<td>5</td>
</tr>
<tr>
<td>00</td>
<td>05</td>
<td>3</td>
</tr>
<tr>
<td>01</td>
<td>02</td>
<td>2</td>
</tr>
<tr>
<td>01</td>
<td>03</td>
<td>3</td>
</tr>
<tr>
<td>01</td>
<td>04</td>
<td>4</td>
</tr>
<tr>
<td>01</td>
<td>06</td>
<td>3</td>
</tr>
<tr>
<td>02</td>
<td>05</td>
<td>7</td>
</tr>
<tr>
<td>02</td>
<td>06</td>
<td>6</td>
</tr>
<tr>
<td>03</td>
<td>07</td>
<td>6</td>
</tr>
<tr>
<td>04</td>
<td>08</td>
<td>10</td>
</tr>
<tr>
<td>04</td>
<td>09</td>
<td>11</td>
</tr>
<tr>
<td>05</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>05</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>06</td>
<td>12</td>
<td>10</td>
</tr>
<tr>
<td>06</td>
<td>13</td>
<td>10</td>
</tr>
<tr>
<td>07</td>
<td>14</td>
<td>8</td>
</tr>
<tr>
<td>07</td>
<td>12</td>
<td>8</td>
</tr>
</tbody>
</table>

**Example 1: Single level explosion**

The first example is called single level explosion. It answers the question, “What parts are needed to build the part identified by ’01’”. The list will include the
direct subparts, subparts of the subparts and so on. However, if a part is used multiple times, its subparts are only listed once.

```sql
WITH RPL (PART, SUBPART, QUANTITY) AS
  ( SELECT ROOT.PART, ROOT.SUBPART, ROOT.QUANTITY
    FROM PARTLIST ROOT
    WHERE ROOT.PART = '01'
    UNION ALL
    SELECT CHILD.PART, CHILD.SUBPART, CHILD.QUANTITY
    FROM RPL PARENT, PARTLIST CHILD
    WHERE PARENT.SUBPART = CHILD.PART
  )
SELECT DISTINCT PART, SUBPART, QUANTITY
FROM RPL
ORDER BY PART, SUBPART, QUANTITY;
```

The above query includes a common table expression, identified by the name `RPL`, that expresses the recursive part of this query. It illustrates the basic elements of a recursive common table expression.

The first operand (full select) of the UNION, referred to as the initialization full select, gets the direct children of part '01'. The FROM clause of this full select refers to the source table and will never refer to itself (`RPL` in this case). The result of this first full select goes into the common table expression `RPL` (Recursive PARTLIST). As in this example, the UNION must always be a UNION ALL.

The second operand (full select) of the UNION uses `RPL` to compute subparts of subparts by having the FROM clause refer to the common table expression `RPL` and the source table with a join of a part from the source table (child) to a subpart of the current result contained in `RPL` (parent). The result goes back to `RPL` again. The second operand of UNION is then used repeatedly until no more children exist.

The SELECT DISTINCT in the main full select of this query ensures the same part/subpart is not listed more than once.

The result of the query is as follows:

<table>
<thead>
<tr>
<th>PART</th>
<th>SUBPART</th>
<th>QUANTITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>02</td>
<td>2</td>
</tr>
<tr>
<td>01</td>
<td>03</td>
<td>3</td>
</tr>
<tr>
<td>01</td>
<td>04</td>
<td>4</td>
</tr>
<tr>
<td>01</td>
<td>06</td>
<td>3</td>
</tr>
<tr>
<td>02</td>
<td>05</td>
<td>7</td>
</tr>
<tr>
<td>02</td>
<td>06</td>
<td>6</td>
</tr>
<tr>
<td>03</td>
<td>07</td>
<td>6</td>
</tr>
<tr>
<td>04</td>
<td>08</td>
<td>10</td>
</tr>
<tr>
<td>04</td>
<td>09</td>
<td>11</td>
</tr>
<tr>
<td>05</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>05</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>06</td>
<td>12</td>
<td>10</td>
</tr>
<tr>
<td>06</td>
<td>13</td>
<td>10</td>
</tr>
<tr>
<td>07</td>
<td>12</td>
<td>8</td>
</tr>
<tr>
<td>07</td>
<td>14</td>
<td>8</td>
</tr>
</tbody>
</table>

Observe in the result that from part '01' we go to '02' which goes to '06' and so on. Further, notice that part '06' is reached twice, once through '01' directly and another time through '02'. In the output, however, its subcomponents are listed only once (this is the result of using a SELECT DISTINCT) as required.
It is important to remember that with recursive common table expressions it is possible to introduce an infinite loop. In this example, an infinite loop would be created if the search condition of the second operand that joins the parent and child tables was coded as:

```
PARENT.SUBPART = CHILD.SUBPART
```

This example of causing an infinite loop is obviously a case of not coding what is intended. However, care should also be exercised in determining what to code so that there is a definite end of the recursion cycle.

The result produced by this example query could be produced in an application program without using a recursive common table expression. However, this approach would require starting of a new query for every level of recursion. Furthermore, the application needs to put all the results back in the database to order the result. This approach complicates the application logic and does not perform well. The application logic becomes even harder and more inefficient for other bill of material queries, such as summarized and indented explosion queries.

**Example 2: Summarized explosion**

The second example is a summarized explosion. The question posed here is, what is the total quantity of each part required to build part '01'. The main difference from the single level explosion is the need to aggregate the quantities. The first example indicates the quantity of subparts required for the part whenever it is required. It does not indicate how many of the subparts are needed to build part '01'.

```
WITH RPL (PART, SUBPART, QUANTITY) AS
(
    SELECT ROOT.PART, ROOT.SUBPART, ROOT.QUANTITY
    FROM PARTLIST ROOT
    WHERE ROOT.PART = '01'
    UNION ALL
    SELECT PARENT.PART, CHILD.SUBPART, PARENT.QUANTITY*CHILD.QUANTITY
    FROM RPL PARENT, PARTLIST CHILD
    WHERE PARENT.SUBPART = CHILD.PART
)
SELECT PART, SUBPART, SUM(QUANTITY) AS "Total Qty Used"
FROM RPL
GROUP BY PART, SUBPART
ORDER BY PART, SUBPART;
```

In the above query, the select list of the second operand of the UNION in the recursive common table expression, identified by the name RPL, shows the aggregation of the quantity. To find out how much of a subpart is used, the quantity of the parent is multiplied by the quantity per parent of a child. If a part is used multiple times in different places, it requires another final aggregation. This is done by the grouping over the common table expression RPL and using the SUM aggregate function in the select list of the main fullselect.

The result of the query is as follows:

<table>
<thead>
<tr>
<th>PART</th>
<th>SUBPART</th>
<th>Total Qty Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>02</td>
<td>2</td>
</tr>
<tr>
<td>01</td>
<td>03</td>
<td>3</td>
</tr>
<tr>
<td>01</td>
<td>04</td>
<td>4</td>
</tr>
<tr>
<td>01</td>
<td>05</td>
<td>14</td>
</tr>
<tr>
<td>01</td>
<td>06</td>
<td>15</td>
</tr>
<tr>
<td>01</td>
<td>07</td>
<td>18</td>
</tr>
<tr>
<td>01</td>
<td>08</td>
<td>40</td>
</tr>
</tbody>
</table>
Looking at the output, consider the line for subpart '06'. The total quantity used value of 15 is derived from a quantity of 3 directly for part '01' and a quantity of 6 for part '02' which is needed 2 times by part '01'.

Example 3: Controlling depth

The question may come to mind, what happens when there are more levels of parts in the table than you are interested in for your query? That is, how is a query written to answer the question, “What are the first two levels of parts needed to build the part identified by '01'?” For the sake of clarity in the example, the level is included in the result.

WITH RPL (LEVEL, PART, SUBPART, QUANTITY) AS

 Sele   1,       ROOT.PART, ROOT.SUBPART, ROOT.QUANTITY
 FRO   PARTLIST ROOT
 WHE   ROOT.PART = '01'
 UNIO   ALL
 SEL   PARENT.LEVEL+1, CHILD.PART, CHILD.SUBPART, CHILD.QUANTITY
 FRO   RPL PARENT, PARTLIST CHILD
 WHE   PARENT.SUBPART = CHILD.PART
 AND   PARENT.LEVEL < 2
 )

 SELECT PART, LEVEL, SUBPART, QUANTITY
 FRO   RPL;

This query is similar to example 1. The column LEVEL was introduced to count the levels from the original part. In the initialization fullselect, the value for the LEVEL column is initialized to 1. In the subsequent fullselect, the level from the parent is incremented by 1. Then to control the number of levels in the result, the second fullselect includes the condition that the parent level must be less than 2. This ensures that the second fullselect only processes children to the second level.

The result of the query is:

<table>
<thead>
<tr>
<th>PART</th>
<th>LEVEL</th>
<th>SUBPART</th>
<th>QUANTITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>1</td>
<td>02</td>
<td>2</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>03</td>
<td>3</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>04</td>
<td>4</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>06</td>
<td>3</td>
</tr>
<tr>
<td>02</td>
<td>2</td>
<td>05</td>
<td>7</td>
</tr>
<tr>
<td>02</td>
<td>2</td>
<td>06</td>
<td>6</td>
</tr>
<tr>
<td>03</td>
<td>2</td>
<td>07</td>
<td>6</td>
</tr>
<tr>
<td>04</td>
<td>2</td>
<td>08</td>
<td>10</td>
</tr>
<tr>
<td>04</td>
<td>2</td>
<td>09</td>
<td>11</td>
</tr>
<tr>
<td>06</td>
<td>2</td>
<td>12</td>
<td>10</td>
</tr>
<tr>
<td>06</td>
<td>2</td>
<td>13</td>
<td>10</td>
</tr>
</tbody>
</table>

update-clause
The FOR UPDATE clause identifies the columns that can be updated in a subsequent Positioned UPDATE statement. Each column-name must be unqualified and must identify a column of the table or view identified in the first FROM clause of the fullselect. If the FOR UPDATE clause is specified without column names, all updatable columns of the table or view identified in the first FROM clause of the fullselect are included.

The FOR UPDATE clause cannot be used if one of the following is true:
- The cursor associated with the select-statement is not deletable.
- One of the selected columns is a non-updatable column of a catalog table and the FOR UPDATE clause has not been used to exclude that column.

The FOR READ ONLY clause indicates that the result table is read-only and therefore the cursor cannot be referred to in Positioned UPDATE and DELETE statements. FOR FETCH ONLY has the same meaning.

Some result tables are read-only by nature. (For example, a table based on a read-only view.) FOR READ ONLY can still be specified for such tables, but the specification has no effect.

For result tables in which updates and deletes are allowed, specifying FOR READ ONLY (or FOR FETCH ONLY) can possibly improve the performance of FETCH operations by allowing the database manager to do blocking. For example, in programs that contain dynamic SQL statements without the FOR READ ONLY or ORDER BY clause, the database manager might open cursors as if the FOR UPDATE clause were specified. It is recommended, therefore, that the FOR READ ONLY clause be used to improve performance, except in cases where queries will be used in positioned UPDATE or DELETE statements.

A read-only result table must not be referred to in a Positioned UPDATE or DELETE statement, whether it is read-only by nature or specified as FOR READ ONLY (FOR FETCH ONLY).

The OPTIMIZE FOR clause requests special processing of the select statement. If the clause is omitted, it is assumed that all rows of the result table will be retrieved; if it is specified, it is assumed that the number of rows retrieved will probably not exceed \( n \), where \( n \) is the value of integer. The value of \( n \) must be a positive integer.

Use of the OPTIMIZE FOR clause influences query optimization, based on the assumption that \( n \) rows will be retrieved. In addition, for cursors that are blocked, this clause will influence the number of rows that will be returned in each block (that is, no more than \( n \) rows will be returned in each block). If both the fetch-first-clause and the optimize-for-clause are specified, the lower of the integer
select-clause

values from these clauses will be used to influence the communications buffer size. The values are considered independently for optimization purposes.

This clause does not limit the number of rows that can be fetched, or affect the result in any other way than performance. Using OPTIMIZE FOR n ROWS can improve performance if no more than n rows are retrieved, but may degrade performance if more than n rows are retrieved.

If the value of n multiplied by the size of the row exceeds the size of the communication buffer, the OPTIMIZE FOR clause will have no impact on the data buffers. The size of the communication buffer is defined by the rqrioblk or the aslheapsz configuration parameter.

isolation-clause

The optional isolation-clause specifies the isolation level at which the statement is executed, and whether a specific type of lock is to be acquired.

- RR - Repeatable Read
- RS - Read Stability
- CS - Cursor Stability
- UR - Uncommitted Read

The default isolation level of the statement is the isolation level of the package in which the statement is bound. When a nickname is used in a select-statement to access data in DB2 family and Microsoft SQL Server data sources, the isolation-clause can be included in the statement to specify the statement isolation level. If the isolation-clause is included in statements that access other data sources, the specified isolation level is ignored. The current isolation level on the federated server is mapped to a corresponding isolation level at the data source on each connection to the data source. After a connection is made to a data source, the isolation level cannot be changed for the duration of the connection.

lock-request-clause

The optional lock-request-clause specifies the type of lock that the database manager is to acquire and hold:

SHARE
Concurrent processes can acquire SHARE or UPDATE locks on the data.

UPDATE
Concurrent processes can acquire SHARE locks on the data, but no concurrent process can acquire an UPDATE or EXCLUSIVE lock.
**EXCLUSIVE**

Concurrent processes cannot acquire a lock on the data.

The lock-request-clause applies to all base table and index scans required by the query, including those within subqueries, SQL functions and SQL methods. It has no affect on locks placed by procedures, external functions, or external methods. Any SQL function or SQL method invoked (directly or indirectly) by the statement must be created with INHERIT ISOLATION LEVEL WITH LOCK REQUEST (SQLSTATE 42601). The lock-request-clause cannot be used with a modifying query that might invoke triggers or that requires referential integrity checks (SQLSTATE 42601).

**Examples of a select-statement**

*Example 1:* Select all columns and rows from the EMPLOYEE table.

```
SELECT * FROM EMPLOYEE
```

*Example 2:* Select the project name (PROJNAME), start date (PRSTDATE), and end date (PRENDATE) from the PROJECT table. Order the result table by the end date with the most recent dates appearing first.

```
SELECT PROJNAME, PRSTDATE, PRENDATE
FROM PROJECT
ORDER BY PRENDATE DESC
```

*Example 3:* Select the department number (WORKDEPT) and average departmental salary (SALARY) for all departments in the EMPLOYEE table. Arrange the result table in ascending order by average departmental salary.

```
SELECT WORKDEPT, AVG(SALARY)
FROM EMPLOYEE
GROUP BY WORKDEPT
ORDER BY 2
```

*Example 4:* Declare a cursor named UP_CUR to be used in a C program to update the start date (PRSTDATE) and the end date (PRENDATE) columns in the PROJECT table. The program must receive both of these values together with the project number (PROJNO) value for each row.

```
EXEC SQL
DECLARE UP_CUR CURSOR FOR
SELECT PROJNO, PRSTDATE, PRENDATE
FROM PROJECT
FOR UPDATE OF PRSTDATE, PRENDATE;
```

*Example 5:* This example names the expression SAL+BONUS+COMM as TOTAL_PAY

```
SELECT SALARY+BONUS+COMM AS TOTAL_PAY
FROM EMPLOYEE
ORDER BY TOTAL_PAY
```

*Example 6:* Determine the employee number and salary of sales representatives along with the average salary and head count of their departments. Also, list the average salary of the department with the highest average salary.

Using a common table expression for this case saves the overhead of creating the DINFO view as a regular view. During statement preparation, accessing the catalog for the view is avoided and, because of the context of the rest of the fullselect, only the rows for the department of the sales representatives need to be considered by the view.
WITH
  DINFO (DEPTNO, AVGSALARY, EMPCOUNT) AS
  (SELECT OTHERS.WORKDEPT, AVG(OTHERS.SALARY), COUNT(*)
    FROM EMPLOYEE OTHERS
    GROUP BY OTHERS.WORKDEPT
  ),
  DINFOMAX AS
  (SELECT MAX(AVGSALARY) AS AVGMAX FROM DINFO)
SELECT THIS_EMP.EMPNO, THIS_EMP.SALARY,
  DINFO.AVGSALARY, DINFO.EMPCOUNT, DINFOMAX.AVGMAX
FROM EMPLOYEE THIS_EMP, DINFO, DINFOMAX
WHERE THIS_EMP.JOB = 'SALESREP'
  AND THIS_EMP.WORKDEPT = DINFO.DEPTNO

Example 7: Given two tables, EMPLOYEE and PROJECT, replace employee SALLY with a new employee GEORGE, assign all projects lead by SALLY to GEORGE, and return the names of the updated projects.

WITH
  NEWEMP AS (SELECT EMPNO FROM NEW TABLE
    (INSERT INTO EMPLOYEE (EMPNO, FIRSTNME)
      VALUES(NEXT VALUE FOR EMPNO_SEQ, 'GEORGE'))),
  OLDEMP AS (SELECT EMPNO FROM EMPLOYEE WHERE FIRSTNME = 'SALLY'),
  UPPROJ AS (SELECT PROJNAME FROM NEW TABLE
    (UPDATE PROJECT
      SET RESPEMP = (SELECT EMPNO FROM NEWEMP)
      WHERE RESPEMP = (SELECT EMPNO FROM OLDEMP))),
  DELEMP AS (SELECT EMPNO FROM OLD TABLE
    (DELETE FROM EMPLOYEE
      WHERE EMPNO = (SELECT EMPNO FROM OLDEMP)))
SELECT PROJNAME FROM UPPROJ;

Example 8: Retrieve data from the DEPT table. That data will later be updated with a searched update, and should be locked when the query executes.

SELECT DEPTNO, DEPTNAME, MGRNO
FROM DEPT
WHERE ADMRDEPT = 'A00'
  FOR READ ONLY WITH RS USE AND KEEP EXCLUSIVE LOCKS
Appendix A. SQL and XML limits

The following tables describe certain SQL and XML limits. Adhering to the most restrictive case can help you to design application programs that are easily portable.

Table 64 lists limits in bytes. These limits are enforced after conversion from the application code page to the database code page when creating identifiers. The limits are also enforced after conversion from the database code page to the application code page when retrieving identifiers from the database. If, during either of these processes, the identifier length limit is exceeded, truncation occurs or an error is returned.

Character limits vary depending on the code page of the database and the code page of the application. For example, because the width of a UTF-8 character can range from 1 to 4 bytes, the character limit for an identifier in a Unicode table whose limit is 128 bytes will range from 32 to 128 characters, depending on which characters are used. If an attempt is made to create an identifier whose name is longer than the limit for this table after conversion to the database code page, an error is returned.

Applications that store identifier names must be able to handle the potentially increased size of identifiers after code page conversion has occurred. When identifiers are retrieved from the catalog, they are converted to the application code page. Conversion from the database code page to the application code page can result in an identifier becoming longer than the byte limit for the table. If a host variable declared by the application cannot store the entire identifier after code page conversion, it is truncated. If that is unacceptable, the host variable can be increased in size to be able to accept the entire identifier name.

The same rules apply to DB2 utilities retrieving data and converting it to a user-specified code page. If a DB2 utility, such as export, is retrieving the data and forcing conversion to a user-specified code page (using the export CODEPAGE modifier or the DB2CODEPAGE registry variable), and the identifier expands beyond the limit that is documented in this table because of code page conversion, an error might be returned or the identifier might be truncated.

Table 64. Identifier Length Limits

<table>
<thead>
<tr>
<th>Description</th>
<th>Maximum in Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alias name</td>
<td>128</td>
</tr>
<tr>
<td>Attribute name</td>
<td>128</td>
</tr>
<tr>
<td>Audit policy name</td>
<td>128</td>
</tr>
<tr>
<td>Authorization name (can only be single-byte characters)</td>
<td>128</td>
</tr>
<tr>
<td>Buffer pool name</td>
<td>18</td>
</tr>
<tr>
<td>Column name $^2$</td>
<td>128</td>
</tr>
<tr>
<td>Constraint name</td>
<td>128</td>
</tr>
<tr>
<td>Correlation name</td>
<td>128</td>
</tr>
<tr>
<td>Cursor name</td>
<td>128</td>
</tr>
<tr>
<td>Data partition name</td>
<td>128</td>
</tr>
</tbody>
</table>
### SQL and XML limits

**Table 64. Identifier Length Limits (continued)**

<table>
<thead>
<tr>
<th>Description</th>
<th>Maximum in Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data source column name</td>
<td>255</td>
</tr>
<tr>
<td>Data source index name</td>
<td>128</td>
</tr>
<tr>
<td>Data source name</td>
<td>128</td>
</tr>
<tr>
<td>Data source table name (remote-table-name)</td>
<td>128</td>
</tr>
<tr>
<td>Database partition group name</td>
<td>128</td>
</tr>
<tr>
<td>Database partition name</td>
<td>128</td>
</tr>
<tr>
<td>Event monitor name</td>
<td>128</td>
</tr>
<tr>
<td>External program name</td>
<td>128</td>
</tr>
<tr>
<td>Function mapping name</td>
<td>128</td>
</tr>
<tr>
<td>Group name</td>
<td>128</td>
</tr>
<tr>
<td>Host identifier(^1)</td>
<td>255</td>
</tr>
<tr>
<td>Identifier for a data source user (remote-authorization-name)</td>
<td>128</td>
</tr>
<tr>
<td>Identifier in an SQL procedure (condition name, for loop identifier, label, result set locator, statement name, variable name)</td>
<td>128</td>
</tr>
<tr>
<td>Index name</td>
<td>128</td>
</tr>
<tr>
<td>Index extension name</td>
<td>18</td>
</tr>
<tr>
<td>Index specification name</td>
<td>128</td>
</tr>
<tr>
<td>Label name</td>
<td>128</td>
</tr>
<tr>
<td>Namespace uniform resource identifier (URI)</td>
<td>1000</td>
</tr>
<tr>
<td>Nickname</td>
<td>128</td>
</tr>
<tr>
<td>Package name</td>
<td>128</td>
</tr>
<tr>
<td>Package version ID</td>
<td>64</td>
</tr>
<tr>
<td>Parameter name</td>
<td>128</td>
</tr>
<tr>
<td>Password to access a data source</td>
<td>32</td>
</tr>
<tr>
<td>Procedure name</td>
<td>128</td>
</tr>
<tr>
<td>Role name</td>
<td>128</td>
</tr>
<tr>
<td>Savepoint name</td>
<td>128</td>
</tr>
<tr>
<td>Schema name(^2)</td>
<td>128</td>
</tr>
<tr>
<td>Security label component name</td>
<td>128</td>
</tr>
<tr>
<td>Security label name</td>
<td>128</td>
</tr>
<tr>
<td>Security policy name</td>
<td>128</td>
</tr>
<tr>
<td>Sequence name</td>
<td>128</td>
</tr>
<tr>
<td>Server (database alias) name</td>
<td>8</td>
</tr>
<tr>
<td>Specific name</td>
<td>128</td>
</tr>
<tr>
<td>SQL condition name</td>
<td>128</td>
</tr>
<tr>
<td>SQL variable name</td>
<td>128</td>
</tr>
<tr>
<td>Statement name</td>
<td>128</td>
</tr>
<tr>
<td>Table name</td>
<td>128</td>
</tr>
<tr>
<td>Table space name</td>
<td>18</td>
</tr>
</tbody>
</table>
## SQL and XML limits

**Table 64. Identifier Length Limits (continued)**

<table>
<thead>
<tr>
<th>Description</th>
<th>Maximum in Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transform group name</td>
<td>18</td>
</tr>
<tr>
<td>Trigger name</td>
<td>128</td>
</tr>
<tr>
<td>Trusted context name</td>
<td>128</td>
</tr>
<tr>
<td>Type mapping name</td>
<td>18</td>
</tr>
<tr>
<td>User-defined function name</td>
<td>128</td>
</tr>
<tr>
<td>User-defined method name</td>
<td>128</td>
</tr>
<tr>
<td>User-defined type name2</td>
<td>128</td>
</tr>
<tr>
<td>View name</td>
<td>128</td>
</tr>
<tr>
<td>Wrapper name</td>
<td>128</td>
</tr>
<tr>
<td>XML element name, attribute name, or prefix name</td>
<td>1000</td>
</tr>
<tr>
<td>XML schema location uniform resource identifier (URI)</td>
<td>1000</td>
</tr>
</tbody>
</table>

**Note:**
1. Individual host language compilers might have a more restrictive limit on variable names.
2. The SQLDA structure is limited to storing 30-byte column names, 18-byte user-defined type names, and 8-byte schema names for user-defined types. Because the SQLDA is used in the DESCRIBE statement, embedded SQL applications that use the DESCRIBE statement to retrieve column or user-defined type name information must conform to these limits.

**Table 65. Numeric Limits**

<table>
<thead>
<tr>
<th>Description</th>
<th>Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smallest SMALLINT value</td>
<td>-32 768</td>
</tr>
<tr>
<td>Largest SMALLINT value</td>
<td>+32 767</td>
</tr>
<tr>
<td>Smallest INTEGER value</td>
<td>-2 147 483 648</td>
</tr>
<tr>
<td>Largest INTEGER value</td>
<td>+2 147 483 647</td>
</tr>
<tr>
<td>Smallest BIGINT value</td>
<td>-9 223 372 036 854 775 808</td>
</tr>
<tr>
<td>Largest BIGINT value</td>
<td>+9 223 372 036 854 775 807</td>
</tr>
<tr>
<td>Largest decimal precision</td>
<td>31</td>
</tr>
<tr>
<td>Maximum exponent (E&lt;sub&gt;max&lt;/sub&gt;) for REAL values</td>
<td></td>
</tr>
<tr>
<td>Smallest REAL value</td>
<td>-3.402E+38</td>
</tr>
<tr>
<td>Largest REAL value</td>
<td>+3.402E+38</td>
</tr>
<tr>
<td>Minimum exponent (E&lt;sub&gt;min&lt;/sub&gt;) for REAL values</td>
<td>-37</td>
</tr>
<tr>
<td>Smallest positive REAL value</td>
<td>+1.175E-37</td>
</tr>
<tr>
<td>Largest negative REAL value</td>
<td>-1.175E-37</td>
</tr>
<tr>
<td>Maximum exponent (E&lt;sub&gt;max&lt;/sub&gt;) for DOUBLE values</td>
<td>308</td>
</tr>
<tr>
<td>Smallest DOUBLE value</td>
<td>-1.79769E+308</td>
</tr>
<tr>
<td>Largest DOUBLE value</td>
<td>+1.79769E+308</td>
</tr>
<tr>
<td>Minimum exponent (E&lt;sub&gt;min&lt;/sub&gt;) for DOUBLE values</td>
<td>-307</td>
</tr>
</tbody>
</table>
### SQL and XML limits

**Table 65. Numeric Limits (continued)**

<table>
<thead>
<tr>
<th>Description</th>
<th>Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smallest positive DOUBLE value</td>
<td>+2.225E-307</td>
</tr>
<tr>
<td>Largest negative DOUBLE value</td>
<td>-2.225E-307</td>
</tr>
<tr>
<td>Maximum exponent ($E_{\text{max}}$) for DECFLOAT(16) values</td>
<td>384</td>
</tr>
<tr>
<td>Smallest DECFLOAT(16) value</td>
<td>-9.9999999999999999E+384</td>
</tr>
<tr>
<td>Largest DECFLOAT(16) value</td>
<td>9.9999999999999999E+384</td>
</tr>
<tr>
<td>Minimum exponent ($E_{\text{min}}$) for DECFLOAT(16) values</td>
<td>-383</td>
</tr>
<tr>
<td>Smallest positive DECFLOAT(16) value</td>
<td>1.0000000000000000E-383</td>
</tr>
<tr>
<td>Largest negative DECFLOAT(16) value</td>
<td>-1.0000000000000000E-383</td>
</tr>
<tr>
<td>Maximum exponent ($E_{\text{max}}$) for DECFLOAT(34) values</td>
<td>6144</td>
</tr>
<tr>
<td>Smallest DECFLOAT(34) value</td>
<td>-9.99999999999999999999999999999999E+6144</td>
</tr>
<tr>
<td>Largest DECFLOAT(34) value</td>
<td>9.99999999999999999999999999999999E+6144</td>
</tr>
<tr>
<td>Minimum exponent ($E_{\text{min}}$) for DECFLOAT(34) values</td>
<td>-6143</td>
</tr>
<tr>
<td>Smallest positive DECFLOAT(34) value</td>
<td>1.0000000000000000000000000000000000E-6143</td>
</tr>
<tr>
<td>Largest negative DECFLOAT(34) value</td>
<td>-1.0000000000000000000000000000000000E-6143</td>
</tr>
</tbody>
</table>

**Note:**

1. These are the limits of normal decimal floating-point numbers. Valid decimal floating-point values include the special values NaN, -NAN, SNAN, -SNAN, INFINITY and -INFINITY. In addition, valid values include subnormal numbers.

   Subnormal numbers are nonzero numbers whose adjusted exponents are less than $E_{\text{min}}$. For a subnormal number, the minimum value of the exponent is $E_{\text{min}} - (\text{precision}-1)$, called $E_{\text{tiny}}$, where \text{precision} is the working precision (16 or 34). That is, subnormal numbers extend the range of numbers close to zero by 15 or 33 orders of magnitude for DECFLOAT(16) or DECFLOAT(34), respectively. Subnormal numbers are different from normal numbers because the maximum number of digits for a subnormal number is less than the working precision (16 or 34). Decimal floating-point cannot represent the subnormal numbers with the same accuracy as it can represent normal numbers. The smallest positive subnormal number for DECFLOAT(34) is $1 \times 10^{-4176}$, which contains only one digit, whereas the smallest positive normal number for DECFLOAT(34) is $1.0000000000000000000000000000000001 \times 10^{-6143}$, which contains 34 digits. The smallest positive subnormal number for DECFLOAT(16) is $1 \times 10^{-308}$.

**Table 66. String Limits**

<table>
<thead>
<tr>
<th>Description</th>
<th>Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum length of CHAR (in bytes)</td>
<td>254</td>
</tr>
<tr>
<td>Maximum length of VARCHAR (in bytes)</td>
<td>32 672</td>
</tr>
<tr>
<td>Maximum length of LONG VARCHAR (in bytes)</td>
<td>32 700</td>
</tr>
</tbody>
</table>
Table 66. String Limits (continued)

<table>
<thead>
<tr>
<th>Description</th>
<th>Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum length of CLOB (in bytes)</td>
<td>2,147,483,647</td>
</tr>
<tr>
<td>Maximum length of serialized XML (in bytes)</td>
<td>2,147,483,647</td>
</tr>
<tr>
<td>Maximum length of GRAPHIC (in double-byte characters)</td>
<td>127</td>
</tr>
<tr>
<td>Maximum length of VARGRAPHIC (in double-byte characters)</td>
<td>16,336</td>
</tr>
<tr>
<td>Maximum length of LONG VARGRAPHIC (in double-byte characters)</td>
<td>16,350</td>
</tr>
<tr>
<td>Maximum length of DBCLOB (in double-byte characters)</td>
<td>1,073,741,823</td>
</tr>
<tr>
<td>Maximum length of BLOB (in bytes)</td>
<td>2,147,483,647</td>
</tr>
<tr>
<td>Maximum length of character constant</td>
<td>32,672</td>
</tr>
<tr>
<td>Maximum length of graphic constant</td>
<td>16,336</td>
</tr>
<tr>
<td>Maximum length of concatenated character string</td>
<td>2,147,483,647</td>
</tr>
<tr>
<td>Maximum length of concatenated graphic string</td>
<td>1,073,741,823</td>
</tr>
<tr>
<td>Maximum length of concatenated binary string</td>
<td>2,147,483,647</td>
</tr>
<tr>
<td>Maximum number of hexadecimal constant digits</td>
<td>32,672</td>
</tr>
<tr>
<td>Largest instance of a structured type column object at run time (in gigabytes)</td>
<td>1</td>
</tr>
<tr>
<td>Maximum size of a catalog comment (in bytes)</td>
<td>254</td>
</tr>
</tbody>
</table>

**Note:**
1. The LONG VARCHAR and LONG VARGRAPHIC data types are deprecated and might be removed in a future release.

Table 67. XML Limits

<table>
<thead>
<tr>
<th>Description</th>
<th>Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum depth of an XML document (in levels)</td>
<td>125</td>
</tr>
<tr>
<td>Maximum size of an XML schema document (in bytes)</td>
<td>31,457,280</td>
</tr>
</tbody>
</table>

Table 68. Datetime Limits

<table>
<thead>
<tr>
<th>Description</th>
<th>Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smallest DATE value</td>
<td>0001-01-01</td>
</tr>
<tr>
<td>Largest DATE value</td>
<td>9999-12-31</td>
</tr>
<tr>
<td>Smallest TIME value</td>
<td>00:00:00</td>
</tr>
<tr>
<td>Largest TIME value</td>
<td>24:00:00</td>
</tr>
<tr>
<td>Smallest TIMESTAMP value</td>
<td>0001-01-01-00:00:00:00000000000000</td>
</tr>
<tr>
<td>Largest TIMESTAMP value</td>
<td>9999-12-31-24:00:00:00000000000000</td>
</tr>
<tr>
<td>Smallest timestamp precision</td>
<td>0</td>
</tr>
<tr>
<td>Largest timestamp precision</td>
<td>12</td>
</tr>
</tbody>
</table>
### SQL and XML limits

**Table 69. Database Manager Limits**

<table>
<thead>
<tr>
<th>Description</th>
<th>Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Applications</strong></td>
<td></td>
</tr>
<tr>
<td>Maximum number of host variable declarations in a precompiled program&lt;sup&gt;3&lt;/sup&gt;</td>
<td>storage</td>
</tr>
<tr>
<td>Maximum length of a host variable value (in bytes)</td>
<td>2 147 483 647</td>
</tr>
<tr>
<td>Maximum number of declared cursors in a program</td>
<td>storage</td>
</tr>
<tr>
<td>Maximum number of rows changed in a unit of work</td>
<td>storage</td>
</tr>
<tr>
<td>Maximum number of cursors opened at one time</td>
<td>storage</td>
</tr>
<tr>
<td>Maximum number of connections per process within a DB2 client</td>
<td>512</td>
</tr>
<tr>
<td>Maximum number of simultaneously opened LOB locators in a transaction</td>
<td>4 194 304</td>
</tr>
<tr>
<td>Maximum size of an SQLDA (in bytes)</td>
<td>storage</td>
</tr>
<tr>
<td>Maximum number of prepared statements</td>
<td>storage</td>
</tr>
<tr>
<td><strong>Buffer Pools</strong></td>
<td></td>
</tr>
<tr>
<td>Maximum NPAGES in a buffer pool for 32-bit releases</td>
<td>1 048 576</td>
</tr>
<tr>
<td>Maximum NPAGES in a buffer pool for 64-bit releases</td>
<td>2 147 483 647</td>
</tr>
<tr>
<td>Maximum total size of all buffer pool slots (4K)</td>
<td>2 147 483 646</td>
</tr>
<tr>
<td><strong>Concurrency</strong></td>
<td></td>
</tr>
<tr>
<td>Maximum number of concurrent users of a server&lt;sup&gt;4&lt;/sup&gt;</td>
<td>64 000</td>
</tr>
<tr>
<td>Maximum number of concurrent users per instance</td>
<td>64 000</td>
</tr>
<tr>
<td>Maximum number of concurrent applications per database</td>
<td>60 000</td>
</tr>
<tr>
<td>Maximum number of databases per instance concurrently in use</td>
<td>256</td>
</tr>
<tr>
<td><strong>Constraints</strong></td>
<td></td>
</tr>
<tr>
<td>Maximum number of constraints on a table</td>
<td>storage</td>
</tr>
<tr>
<td>Maximum number of columns in a UNIQUE constraint (supported through a UNIQUE index)</td>
<td>64</td>
</tr>
<tr>
<td>Maximum combined length of columns in a UNIQUE constraint (supported through a UNIQUE index, in bytes)&lt;sup&gt;5&lt;/sup&gt;</td>
<td>8192</td>
</tr>
<tr>
<td>Maximum number of referencing columns in a foreign key</td>
<td>64</td>
</tr>
<tr>
<td>Maximum combined length of referencing columns in a foreign key&lt;sup&gt;6&lt;/sup&gt;</td>
<td>8192</td>
</tr>
<tr>
<td>Maximum length of a check constraint specification (in bytes)</td>
<td>65 535</td>
</tr>
<tr>
<td><strong>Databases</strong></td>
<td></td>
</tr>
<tr>
<td>Maximum database partition number</td>
<td>999</td>
</tr>
<tr>
<td><strong>Indexes</strong></td>
<td></td>
</tr>
<tr>
<td>Maximum number of indexes on a table</td>
<td>32 767 or storage</td>
</tr>
<tr>
<td>Maximum number of columns in an index key</td>
<td>64</td>
</tr>
<tr>
<td>Maximum length of an index key including all overhead&lt;sup&gt;7&lt;/sup&gt;&lt;sub&gt;9&lt;/sub&gt;</td>
<td>indexpagesize/4</td>
</tr>
<tr>
<td>Maximum length of a variable index key part (in bytes)&lt;sup&gt;8&lt;/sup&gt;</td>
<td>1022 or storage</td>
</tr>
<tr>
<td>Maximum size of an index per database partition in an SMS table space (in gigabytes)&lt;sup&gt;7&lt;/sup&gt;</td>
<td>16 384</td>
</tr>
</tbody>
</table>
### SQL and XML limits

**Table 69. Database Manager Limits (continued)**

<table>
<thead>
<tr>
<th>Description</th>
<th>Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum size of an index per database partition in a regular DMS table space (in gigabytes)</td>
<td>512</td>
</tr>
<tr>
<td>Maximum size of an index per database partition in a large DMS table space (in gigabytes)</td>
<td>16 384</td>
</tr>
<tr>
<td>Maximum size of an index over XML data per database partition (in terabytes)</td>
<td>2</td>
</tr>
<tr>
<td>Maximum length of a variable index key part for an index over XML data (in bytes)</td>
<td>pagesize/4 - 207</td>
</tr>
</tbody>
</table>

**Log records**

| Maximum Log Sequence Number | 17 984 000 000 000 000 000 |

**Monitoring**

<table>
<thead>
<tr>
<th>Maximum number of simultaneously active event monitors</th>
<th>128</th>
</tr>
</thead>
<tbody>
<tr>
<td>With DB2 Database Partitioning Feature (DPF), maximum number of simultaneously active GLOBAL event monitors</td>
<td>32</td>
</tr>
</tbody>
</table>

**Routines**

<table>
<thead>
<tr>
<th>Maximum number of parameters in a procedure</th>
<th>32 767</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum number of parameters in a cursor value constructor</td>
<td>32 767</td>
</tr>
<tr>
<td>Maximum number of parameters in a user-defined function</td>
<td>90</td>
</tr>
<tr>
<td>Maximum number of nested levels for routines</td>
<td>64</td>
</tr>
<tr>
<td>Maximum number of schemas in the SQL path</td>
<td>64</td>
</tr>
<tr>
<td>Maximum length of the SQL path (in bytes)</td>
<td>2048</td>
</tr>
</tbody>
</table>

**Security**

| Maximum number of elements in a security label component of type set or tree | 64                           |
| Maximum number of elements in a security label component of type array | 65 535                       |
| Maximum number of security label components in a security policy | 16                           |

**SQL**

<table>
<thead>
<tr>
<th>Maximum total length of an SQL statement (in bytes)</th>
<th>2 097 152</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum number of tables referenced in an SQL statement or a view</td>
<td>storage</td>
</tr>
<tr>
<td>Maximum number of host variable references in an SQL statement</td>
<td>32 767</td>
</tr>
<tr>
<td>Maximum number of constants in a statement</td>
<td>storage</td>
</tr>
<tr>
<td>Maximum number of elements in a select list</td>
<td>1012</td>
</tr>
<tr>
<td>Maximum number of predicates in a WHERE or HAVING clause</td>
<td>storage</td>
</tr>
<tr>
<td>Maximum number of columns in a GROUP BY clause</td>
<td>1012</td>
</tr>
<tr>
<td>Maximum total length of columns in a GROUP BY clause (in bytes)</td>
<td>32 677</td>
</tr>
<tr>
<td>Maximum number of columns in an ORDER BY clause</td>
<td>1012</td>
</tr>
</tbody>
</table>
### SQL and XML limits

#### Table 69. Database Manager Limits (continued)

<table>
<thead>
<tr>
<th>Description</th>
<th>Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum total length of columns in an ORDER BY clause (in bytes)</td>
<td>32,677</td>
</tr>
<tr>
<td>Maximum level of subquery nesting</td>
<td>storage</td>
</tr>
<tr>
<td>Maximum number of subqueries in a single statement</td>
<td>storage</td>
</tr>
<tr>
<td>Maximum number of values in an insert operation</td>
<td>1012</td>
</tr>
<tr>
<td>Maximum number of SET clauses in a single update operation</td>
<td>1012</td>
</tr>
<tr>
<td><strong>Tables and Views</strong></td>
<td></td>
</tr>
<tr>
<td>Maximum number of columns in a table</td>
<td>1012</td>
</tr>
<tr>
<td>Maximum number of columns in a view</td>
<td>5000</td>
</tr>
<tr>
<td>Maximum number of columns in a data source table or view that is referenced by a nickname</td>
<td>5000</td>
</tr>
<tr>
<td>Maximum number of columns in a distribution key</td>
<td>500</td>
</tr>
<tr>
<td>Maximum length of a row including all overhead</td>
<td>32,677</td>
</tr>
<tr>
<td>Maximum number of rows in a non-partitioned table, per database partition</td>
<td>128 x 10¹⁰</td>
</tr>
<tr>
<td>Maximum number of rows in a data partition, per database partition</td>
<td>128 x 10¹⁰</td>
</tr>
<tr>
<td>Maximum size of a table per database partition in a regular table space (in gigabytes)</td>
<td>512</td>
</tr>
<tr>
<td>Maximum size of a table per database partition in a large DMS table space (in gigabytes)</td>
<td>16,384</td>
</tr>
<tr>
<td>Maximum number of data partitions for a single table</td>
<td>32,767</td>
</tr>
<tr>
<td>Maximum number of table partitioning columns</td>
<td>16</td>
</tr>
<tr>
<td>Maximum number of fields in a user-defined row type</td>
<td>1012</td>
</tr>
<tr>
<td><strong>Table Spaces</strong></td>
<td></td>
</tr>
<tr>
<td>Maximum size of a LOB object (in terabytes)</td>
<td>4</td>
</tr>
<tr>
<td>Maximum size of a LF object (in terabytes)</td>
<td>2</td>
</tr>
<tr>
<td>Maximum number of table spaces in a database</td>
<td>32,768</td>
</tr>
<tr>
<td>Maximum number of tables in an SMS table space</td>
<td>65,534</td>
</tr>
<tr>
<td>Maximum size of a regular DMS table space (in gigabytes)</td>
<td>512</td>
</tr>
<tr>
<td>Maximum size of a large DMS table space (in terabytes)</td>
<td>64</td>
</tr>
<tr>
<td>Maximum size of a temporary DMS table space (in terabytes)</td>
<td>64</td>
</tr>
<tr>
<td>Maximum number of table objects in a DMS table space</td>
<td>51,000</td>
</tr>
<tr>
<td>Maximum number of storage paths in an automatic storage database</td>
<td>128</td>
</tr>
<tr>
<td>Maximum length of a storage path that is associated with an automatic storage database (in bytes)</td>
<td>175</td>
</tr>
<tr>
<td><strong>Triggers</strong></td>
<td></td>
</tr>
<tr>
<td>Maximum run-time depth of cascading triggers</td>
<td>16</td>
</tr>
<tr>
<td><strong>User-defined Types</strong></td>
<td></td>
</tr>
</tbody>
</table>
### SQL and XML limits

**Table 69. Database Manager Limits (continued)**

<table>
<thead>
<tr>
<th>Description</th>
<th>Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum number of attributes in a structured type</td>
<td>4082</td>
</tr>
</tbody>
</table>

**Note:**

1. This maximum can be achieved using a join in the CREATE VIEW statement. Selecting from such a view is subject to the limit of most elements in a select list.
2. The actual data for BLOB, CLOB, LONG VARCHAR, DBCLOB, and LONG VARGRAPHIC columns is not included in this count. However, information about the location of that data does take up some space in the row.
3. The numbers shown are architectural limits and approximations. The practical limits may be less.
4. The actual value is controlled by the `max_connections` and `max_coordagents` database manager configuration parameters.
5. This is an architectural limit. The limit on the most columns in an index key should be used as the practical limit.
6. Table objects include data, indexes, LONG VARCHAR or VARGRAPHIC columns, and LOB columns. Table objects that are in the same table space as the table data do not count extra toward the limit. However, each table object that is in a different table space than the table data does contribute one toward the limit for each table object type per table in the table space in which the table object resides.
7. For page size-specific values, see Table 70.
8. This is limited only by the longest index key, including all overhead (in bytes). As the number of index key parts increases, the maximum length of each key part decreases.
9. The maximum can be less, depending on index options.

<table>
<thead>
<tr>
<th>Description</th>
<th>4K page size limit</th>
<th>8K page size limit</th>
<th>16K page size limit</th>
<th>32K page size limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum number of columns in a table</td>
<td>500</td>
<td>1012</td>
<td>1012</td>
<td>1012</td>
</tr>
<tr>
<td>Maximum length of a row including all overhead</td>
<td>4005</td>
<td>8101</td>
<td>16 293</td>
<td>32 677</td>
</tr>
<tr>
<td>Maximum size of a table per database partition in a regular table space (in gigabytes)</td>
<td>64</td>
<td>128</td>
<td>256</td>
<td>512</td>
</tr>
<tr>
<td>Maximum size of a table per database partition in a large DMS table space (in gigabytes)</td>
<td>2048</td>
<td>4096</td>
<td>8192</td>
<td>16 384</td>
</tr>
<tr>
<td>Maximum length of an index key including all overhead (in bytes)</td>
<td>1024</td>
<td>2048</td>
<td>4096</td>
<td>8192</td>
</tr>
<tr>
<td>Maximum size of an index per database partition in an SMS table space (in gigabytes)</td>
<td>2048</td>
<td>4096</td>
<td>8192</td>
<td>16 384</td>
</tr>
<tr>
<td>Maximum size of an index per database partition in a regular DMS table space (in gigabytes)</td>
<td>64</td>
<td>128</td>
<td>256</td>
<td>512</td>
</tr>
</tbody>
</table>
### SQL and XML limits

Table 70. Database Manager Page Size-specific Limits (continued)

<table>
<thead>
<tr>
<th>Description</th>
<th>4K page size limit</th>
<th>8K page size limit</th>
<th>16K page size limit</th>
<th>32K page size limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum size of an index per database partition in a large DMS table space (in gigabytes)</td>
<td>2048</td>
<td>4096</td>
<td>8192</td>
<td>16 384</td>
</tr>
<tr>
<td>Maximum size of an index over XML data per database partition (in terabytes)</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Maximum size of a regular DMS table space (in gigabytes)</td>
<td>64</td>
<td>128</td>
<td>256</td>
<td>512</td>
</tr>
<tr>
<td>Maximum size of a large DMS table space (in terabytes)</td>
<td>8</td>
<td>16</td>
<td>32</td>
<td>64</td>
</tr>
<tr>
<td>Maximum size of a temporary DMS table space (in terabytes)</td>
<td>8</td>
<td>16</td>
<td>32</td>
<td>64</td>
</tr>
<tr>
<td>Maximum number of elements in a select list</td>
<td>500</td>
<td>1012</td>
<td>1012</td>
<td>1012</td>
</tr>
<tr>
<td>Maximum number of columns in a GROUP BY clause</td>
<td>500</td>
<td>1012</td>
<td>1012</td>
<td>1012</td>
</tr>
<tr>
<td>Maximum total length of columns in a GROUP BY clause (in bytes)</td>
<td>4005</td>
<td>8101</td>
<td>16 293</td>
<td>32 677</td>
</tr>
<tr>
<td>Maximum number of columns in an ORDER BY clause</td>
<td>500</td>
<td>1012</td>
<td>1012</td>
<td>1012</td>
</tr>
<tr>
<td>Maximum total length of columns in an ORDER BY clause (in bytes)</td>
<td>4005</td>
<td>8101</td>
<td>16 293</td>
<td>32 677</td>
</tr>
<tr>
<td>Maximum number of values in an insert operation</td>
<td>500</td>
<td>1012</td>
<td>1012</td>
<td>1012</td>
</tr>
<tr>
<td>Maximum number of SET clauses in a single update operation</td>
<td>500</td>
<td>1012</td>
<td>1012</td>
<td>1012</td>
</tr>
<tr>
<td>Maximum records per page for a regular table space</td>
<td>251</td>
<td>253</td>
<td>254</td>
<td>253</td>
</tr>
<tr>
<td>Maximum records per page for a large table space</td>
<td>287</td>
<td>580</td>
<td>1165</td>
<td>2335</td>
</tr>
</tbody>
</table>
Appendix B. SQLCA (SQL communications area)

An SQLCA is a collection of variables that is updated at the end of the execution of every SQL statement. A program that contains executable SQL statements and is precompiled with option LANGLEVEL SAA1 (the default) or MIA must provide exactly one SQLCA, though more than one SQLCA is possible by having one SQLCA per thread in a multi-threaded application.

When a program is precompiled with option LANGLEVEL SQL92E, an SQLCODE or SQLSTATE variable may be declared in the SQL declare section or an SQLCODE variable can be declared somewhere in the program.

An SQLCA should not be provided when using LANGLEVEL SQL92E. The SQL INCLUDE statement can be used to provide the declaration of the SQLCA in all languages but REXX. The SQLCA is automatically provided in REXX.

To display the SQLCA after each command executed through the command line processor, issue the command db2 -a. The SQLCA is then provided as part of the output for subsequent commands. The SQLCA is also dumped in the db2diag log file.

SQLCA field descriptions

Table 71. Fields of the SQLCA. The field names shown are those present in an SQLCA that is obtained via an INCLUDE statement.

<table>
<thead>
<tr>
<th>Name</th>
<th>Data Type</th>
<th>Field Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>sqlcaid</td>
<td>CHAR(8)</td>
<td>An &quot;eye catcher” for storage dumps containing 'SQLCA’. The sixth byte is ‘L’ if line number information is returned from parsing an SQL procedure body.</td>
</tr>
<tr>
<td>sqlcabc</td>
<td>INTEGER</td>
<td>Contains the length of the SQLCA, 136.</td>
</tr>
<tr>
<td>sqlcode</td>
<td>INTEGER</td>
<td>Contains the SQL return code.</td>
</tr>
<tr>
<td>Code</td>
<td>Means</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Successful execution (although one or more SQLWARN indicators may be set).</td>
<td></td>
</tr>
<tr>
<td>positive</td>
<td>Successful execution, but with a warning condition.</td>
<td></td>
</tr>
<tr>
<td>negative</td>
<td>Error condition.</td>
<td></td>
</tr>
<tr>
<td>sqlerrml</td>
<td>SMALLINT</td>
<td>Length indicator for sqlerrmc, in the range 0 through 70. 0 means that the value of sqlerrmc is not relevant.</td>
</tr>
</tbody>
</table>
Table 71. Fields of the SQLCA (continued). The field names shown are those present in an SQLCA that is obtained via an INCLUDE statement.

<table>
<thead>
<tr>
<th>Name</th>
<th>Data Type</th>
<th>Field Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>sqlerrmc</td>
<td>VARCHAR</td>
<td>Contains one or more tokens, separated by X'FF', which are</td>
</tr>
<tr>
<td></td>
<td></td>
<td>substituted for variables in the descriptions of error conditions.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This field is also used when a successful connection is completed.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>When a NOT ATOMIC compound SQL statement is issued, it may contain information on up to seven errors. The last token might be followed by X'FF'. The sqlerrml value will include any trailing X'FF'.</td>
</tr>
<tr>
<td>sqlerrp</td>
<td>CHAR(8)</td>
<td>Begins with a three-letter identifier indicating the product, followed by five alphanumeric characters indicating the version, release, and modification level of the product. The characters A-Z indicate a modification level higher than 9. A indicates modification level 10, B indicates modification level 11, and so on. For example, SQL0907C means DB2 Version 9, release 7, modification level 12).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If SQLCODE indicates an error condition, this field identifies the module that returned the error. This field is also used when a successful connection is completed.</td>
</tr>
<tr>
<td>sqlerrd</td>
<td>ARRAY</td>
<td>Six INTEGER variables that provide diagnostic information. These values are generally empty if there are no errors, except for sqlerrd(6) from a partitioned database.</td>
</tr>
<tr>
<td>sqlerrd(1)</td>
<td>INTEGER</td>
<td>If connection is invoked and successful, contains the maximum expected difference in length of mixed character data (CHAR data types) when converted to the database code page from the application code page. A value of 0 or 1 indicates no expansion; a value greater than 1 indicates a possible expansion in length; a negative value indicates a possible contraction. On successful return from an SQL procedure, contains the return status value from the SQL procedure.</td>
</tr>
<tr>
<td>sqlerrd(2)</td>
<td>INTEGER</td>
<td>If connection is invoked and successful, contains the maximum expected difference in length of mixed character data (CHAR data types) when converted to the application code page from the database code page. A value of 0 or 1 indicates no expansion; a value greater than 1 indicates a possible expansion in length; a negative value indicates a possible contraction. If the SQLCA results from a NOT ATOMIC compound SQL statement that encountered one or more errors, the value is set to the number of statements that failed.</td>
</tr>
</tbody>
</table>
### SQLCA (SQL communications area)

**Table 71. Fields of the SQLCA (continued).** The field names shown are those present in an SQLCA that is obtained via an INCLUDE statement.

<table>
<thead>
<tr>
<th>Name</th>
<th>Data Type</th>
<th>Field Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>sqlerrd(3)</td>
<td>INTEGER</td>
<td>If PREPARE is invoked and successful, contains an estimate of the number of rows that will be returned. After INSERT, UPDATE, DELETE, or MERGE, contains the actual number of rows that qualified for the operation. For a TRUNCATE statement, the value will be -1. If compound SQL is invoked, contains an accumulation of all sub-statement rows. If CONNECT is invoked, contains 1 if the database can be updated, or 2 if the database is read only. If the OPEN statement is invoked, and the cursor contains SQL data change statements, this field contains the sum of the number of rows that qualified for the embedded insert, update, delete, or merge operations. If CREATE PROCEDURE for an SQL procedure is invoked, and an error is encountered when parsing the SQL procedure body, contains the line number where the error was encountered. The sixth byte of sqlcaid must be 'L' for this to be a valid line number.</td>
</tr>
<tr>
<td>sqlerrd(4)</td>
<td>INTEGER</td>
<td>If PREPARE is invoked and successful, contains a relative cost estimate of the resources required to process the statement. If compound SQL is invoked, contains a count of the number of successful sub-statements. If CONNECT is invoked, contains 0 for a one-phase commit from a down-level client; 1 for a one-phase commit; 2 for a one-phase, read-only commit; and 3 for a two-phase commit.</td>
</tr>
</tbody>
</table>
| sqlerrd(5) | INTEGER   | Contains the total number of rows deleted, inserted, or updated as a result of both:  
  - The enforcement of constraints after a successful delete operation  
  - The processing of triggered SQL statements from activated triggers  
  If compound SQL is invoked, contains an accumulation of the number of such rows for all sub-statements. In some cases, when an error is encountered, this field contains a negative value that is an internal error pointer. If CONNECT is invoked, contains an authentication type value of 0 for server authentication; 1 for client authentication; 2 for authentication using DB2 Connect; 4 for SERVER_ENCRYPT authentication; 5 for authentication using DB2 Connect with encryption; 7 for KERBEROS authentication; 9 for GSSPLUGIN authentication; 11 for DATA_ENCRYPT authentication; and 255 for unspecified authentication. |
| sqlerrd(6) | INTEGER   | For a partitioned database, contains the partition number of the database partition that encountered the error or warning. If no errors or warnings were encountered, this field contains the partition number of the coordinator partition. The number in this field is the same as that specified for the database partition in the db2nodes.cfg file. |
Table 71. Fields of the SQLCA (continued). The field names shown are those present in an SQLCA that is obtained via an INCLUDE statement.

<table>
<thead>
<tr>
<th>Name</th>
<th>Data Type</th>
<th>Field Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>sqlwarn</td>
<td>Array</td>
<td>A set of warning indicators, each containing a blank or W. If compound SQL is invoked, contains an accumulation of the warning indicators set for all sub-statements.</td>
</tr>
<tr>
<td>sqlwarn0</td>
<td>CHAR(1)</td>
<td>Blank if all other indicators are blank; contains ‘W’ if at least one other indicator is not blank.</td>
</tr>
<tr>
<td>sqlwarn1</td>
<td>CHAR(1)</td>
<td>Contains ‘W’ if the value of a string column was truncated when assigned to a host variable. Contains ‘N’ if the null terminator was truncated. Contains ‘A’ if the CONNECT or ATTACH is successful, and the authorization name for the connection is longer than 8 bytes. Contains ‘P’ if the PREPARE statement relative cost estimate stored in sqlerrd(4) exceeded the value that could be stored in an INTEGER or was less than 1, and either the CURRENT EXPLAIN MODE or the CURRENT EXPLAIN SNAPSHOT special register is set to a value other than NO.</td>
</tr>
<tr>
<td>sqlwarn2</td>
<td>CHAR(1)</td>
<td>Contains ‘W’ if null values were eliminated from the argument of an aggregate function. If CONNECT is invoked and successful, contains ‘D’ if the database is in quiesce state, or ‘I’ if the instance is in quiesce state.</td>
</tr>
<tr>
<td>sqlwarn3</td>
<td>CHAR(1)</td>
<td>Contains ‘W’ if the number of columns is not equal to the number of host variables. Contains ‘Z’ if the number of result set locators specified on the ASSOCIATE LOCATORS statement is less than the number of result sets returned by a procedure.</td>
</tr>
<tr>
<td>sqlwarn4</td>
<td>CHAR(1)</td>
<td>Contains ‘W’ if a prepared UPDATE or DELETE statement does not include a WHERE clause.</td>
</tr>
<tr>
<td>sqlwarn5</td>
<td>CHAR(1)</td>
<td>Contains ‘E’ if an error was tolerated during SQL statement execution.</td>
</tr>
<tr>
<td>sqlwarn6</td>
<td>CHAR(1)</td>
<td>Contains ‘W’ if the result of a date calculation was adjusted to avoid an impossible date.</td>
</tr>
<tr>
<td>sqlwarn7</td>
<td>CHAR(1)</td>
<td>Reserved for future use. If CONNECT is invoked and successful, contains ‘E’ if the dyn_query_mgmt database configuration parameter is enabled.</td>
</tr>
<tr>
<td>sqlwarn8</td>
<td>CHAR(1)</td>
<td>Contains ‘W’ if a character that could not be converted was replaced with a substitution character. Contains ‘Y’ if there was an unsuccessful attempt to establish a trusted connection.</td>
</tr>
<tr>
<td>sqlwarn9</td>
<td>CHAR(1)</td>
<td>Contains ‘W’ if arithmetic expressions with errors were ignored during aggregate function processing.</td>
</tr>
<tr>
<td>sqlwarn10</td>
<td>CHAR(1)</td>
<td>Contains ‘W’ if there was a conversion error when converting a character data value in one of the fields in the SQLCA.</td>
</tr>
<tr>
<td>sqlstate</td>
<td>CHAR(5)</td>
<td>A return code that indicates the outcome of the most recently executed SQL statement.</td>
</tr>
</tbody>
</table>

* Some functions may not set SQLWARN2 to W, even though null values were eliminated, because the result was not dependent on the elimination of null values.
Error reporting

The order of error reporting is as follows:

1. Severe error conditions are always reported. When a severe error is reported, there are no additions to the SQLCA.
2. If no severe error occurs, a deadlock error takes precedence over other errors.
3. For all other errors, the SQLCA for the first negative SQL code is returned.
4. If no negative SQL codes are detected, the SQLCA for the first warning (that is, positive SQL code) is returned.

In a partitioned database system, the exception to this rule occurs if a data manipulation operation is invoked against a table that is empty on one database partition, but has data on other database partitions. SQLCODE +100 is only returned to the application if agents from all database partitions return SQL0100W, either because the table is empty on all database partitions, or there are no more rows that satisfy the WHERE clause in an UPDATE statement.

SQLCA usage in partitioned database systems

In partitioned database systems, one SQL statement may be executed by a number of agents on different database partitions, and each agent may return a different SQLCA for different errors or warnings. The coordinator agent also has its own SQLCA.

To provide a consistent view for applications, all SQLCA values are merged into one structure, and SQLCA fields indicate global counts, such that:

- For all errors and warnings, the sqlwarn field contains the warning flags received from all agents.
- Values in the sqlerrd fields indicating row counts are accumulations from all agents.

Note that SQLSTATE 09000 may not be returned every time an error occurs during the processing of a triggered SQL statement.
Appendix C. SQLDA (SQL descriptor area)

An SQLDA is a collection of variables that is required for execution of the SQL DESCRIBE statement. The SQLDA variables are options that can be used by the PREPARE, OPEN, FETCH, and EXECUTE statements. An SQLDA communicates with dynamic SQL; it can be used in a DESCRIBE statement, modified with the addresses of host variables, and then reused in a FETCH or EXECUTE statement.

SQLDAs are supported for all languages, but predefined declarations are provided only for C, REXX, FORTRAN, and COBOL.

The meaning of the information in an SQLDA depends on its use. In PREPARE and DESCRIBE, an SQLDA provides information to an application program about a prepared statement. In OPEN, EXECUTE, and FETCH, an SQLDA describes host variables.

In DESCRIBE and PREPARE, if any one of the columns being described is either a LOB type (LOB locators and file reference variables do not require doubled SQLDAs), reference type, or a user-defined type, the number of SQLVAR entries for the entire SQLDA will be doubled. For example:
- When describing a table with 3 VARCHAR columns and 1 INTEGER column, there will be 4 SQLVAR entries
- When describing a table with 2 VARCHAR columns, 1 CLOB column, and 1 integer column, there will be 8 SQLVAR entries

In EXECUTE, FETCH, and OPEN, if any one of the variables being described is a LOB type (LOB locators and file reference variables do not require doubled SQLDAs) or a structured type, the number of SQLVAR entries for the entire SQLDA must be doubled. (Distinct types and reference types are not relevant in these cases, because the additional information in the double entries is not required by the database. Array, cursor and row types are not supported as SQLDA variables in EXECUTE, FETCH and OPEN statements.)

SQLDA field descriptions

An SQLDA consists of four variables followed by an arbitrary number of occurrences of a sequence of variables collectively named SQLVAR. In OPEN, FETCH, and EXECUTE, each occurrence of SQLVAR describes a host variable. In DESCRIBE and PREPARE, each occurrence of SQLVAR describes a column of a result table or a parameter marker. There are two types of SQLVAR entries:
- **Base SQLVARs**: These entries are always present. They contain the base information about the column, parameter marker, or host variable such as data type code, length attribute, column name, host variable address, and indicator variable address.
- **Secondary SQLVARs**: These entries are only present if the number of SQLVAR entries is doubled as per the rules outlined above. For user-defined types (excluding reference types), they contain the user-defined type name. For reference types, they contain the target type of the reference. For LOBs, they contain the length attribute of the host variable and a pointer to the buffer that contains the actual length. (The distinct type and LOB information does not overlap, so distinct types can be based on LOBs without forcing the number of
SQLDA (SQL descriptor area)

SQLVAR entries on a DESCRIBE to be tripled.) If locators or file reference variables are used to represent LOBs, these entries are not necessary.

In SQLDAs that contain both types of entries, the base SQLVARs are in a block before the block of secondary SQLVARs. In each, the number of entries is equal to the value in SQLD (even though many of the secondary SQLVAR entries may be unused).

The circumstances under which the SQLVAR entries are set by DESCRIBE is detailed in "Effect of DESCRIBE on the SQLDA" on page 766.

Fields in the SQLDA header

Table 72. Fields in the SQLDA Header

<table>
<thead>
<tr>
<th>C Name</th>
<th>SQL Data Type</th>
<th>Usage in DESCRIBE and PREPARE (set by the database manager except for SQLN)</th>
<th>Usage in FETCH, OPEN, and EXECUTE (set by the application prior to executing the statement)</th>
</tr>
</thead>
<tbody>
<tr>
<td>sqldaid</td>
<td>CHAR(8)</td>
<td>The seventh byte of this field is a flag byte named SQLDOUBLED. The database manager sets SQLDOUBLED to the character '2' if two SQLVAR entries have been created for each column; otherwise it is set to a blank (X'20' in ASCII, X'40' in EBCDIC). See &quot;Effect of DESCRIBE on the SQLDA&quot; on page 766 for details on when SQLDOUBLED is set.</td>
<td>The seventh byte of this field is used when the number of SQLVARs is doubled. It is named SQLDOUBLED. If any of the host variables being described is a structured type, BLOB, CLOB, or DBCLOB, the seventh byte must be set to the character '2'; otherwise it can be set to any character but the use of a blank is recommended.</td>
</tr>
<tr>
<td>sqldabc</td>
<td>INTEGER</td>
<td>For 32 bit, the length of the SQLDA, equal to SQLN<em>44+16. For 64 bit, the length of the SQLDA, equal to SQLN</em>56+16</td>
<td>For 32 bit, the length of the SQLDA, &gt;= to SQLN<em>44+16. For 64 bit, the length of the SQLDA, &gt;= to SQLN</em>56+16.</td>
</tr>
<tr>
<td>sqln</td>
<td>SMALLINT</td>
<td>Unchanged by the database manager. Must be set to a value greater than or equal to zero before the DESCRIBE statement is executed. Indicates the total number of occurrences of SQLVAR.</td>
<td>Total number of occurrences of SQLVAR provided in the SQLDA. SQLN must be set to a value greater than or equal to zero.</td>
</tr>
<tr>
<td>sqld</td>
<td>SMALLINT</td>
<td>Set by the database manager to the number of columns in the result table or to the number of parameter markers.</td>
<td>The number of host variables described by occurrences of SQLVAR.</td>
</tr>
</tbody>
</table>
**Fields in an occurrence of a base SQLVAR**

<table>
<thead>
<tr>
<th>Name</th>
<th>Data Type</th>
<th>Usage in DESCRIBE and PREPARE</th>
<th>Usage in FETCH, OPEN, and EXECUTE</th>
</tr>
</thead>
<tbody>
<tr>
<td>sqltype</td>
<td>SMALLINT</td>
<td>Indicates the data type of the column or parameter marker, and whether it can contain nulls. (Parameter markers are always considered nullable.) <a href="#">Table 75 on page 767</a> lists the allowable values and their meanings.</td>
<td>Same for host variable. Host variables for datetime values must be character string variables. For FETCH, a datetime type code means a fixed-length character string. If sqltype is an even number value, the sqlind field is ignored.</td>
</tr>
<tr>
<td>sqllen</td>
<td>SMALLINT</td>
<td>The length attribute of the column or parameter marker. For datetime columns and parameter markers, the length of the string representation of the values. See <a href="#">Table 75 on page 767</a></td>
<td>The length attribute of the host variable. See <a href="#">Table 75 on page 767</a> Note that the value is ignored by the database manager for CLOB, DBCLOB, and BLOB columns. The len.sqllonglen field in the Secondary SQLVAR is used instead.</td>
</tr>
<tr>
<td>sqldata</td>
<td>pointer</td>
<td>For string SQLVARS, sqldata contains the code page. For character-string SQLVARS where the column is defined with the FOR BIT DATA attribute, sqldata contains 0. For other character-string SQLVARS, sqldata contains either the SBCS code page for SBCS data, or the SBCS code page associated with the composite MBCS code page for MBCS data. For Japanese EUC, Traditional Chinese EUC, and Unicode UTF-8 character-string SQLVARS, sqldata contains 954, 964, and 1208 respectively. For all other column types, sqldata is undefined.</td>
<td>Contains the address of the host variable (where the fetched data will be stored).</td>
</tr>
<tr>
<td>sqlind</td>
<td>pointer</td>
<td>For character-string SQLVARS, sqlind contains 0, except for MBCS data, when sqlind contains the DBCS code page associated with the composite MBCS code page.</td>
<td>Contains the address of an associated indicator variable, if there is one; otherwise, not used. If sqltype is an even number value, the sqlind field is ignored.</td>
</tr>
</tbody>
</table>

Note that for a distinct, array, cursor, row, or reference type, the data type of the base type is placed into this field. For a structured type, the data type of the result of the FROM SQL transform function of the transform group (based on the CURRENT DEFAULT TRANSFORM GROUP special register) for the type is placed into this field. There is no indication in the base SQLVAR that it is part of the description of a user-defined type or reference type.
### SQLDA (SQL descriptor area)

**Table 73. Fields in a Base SQLVAR (continued)**

<table>
<thead>
<tr>
<th>Name</th>
<th>Data Type</th>
<th>Usage in DESCRIBE and PREPARE</th>
<th>Usage in FETCH, OPEN, and EXECUTE</th>
</tr>
</thead>
<tbody>
<tr>
<td>sqlname</td>
<td>VARCHAR</td>
<td>Contains the unqualified name of the column or parameter marker.</td>
<td>When connecting to a host database, sqlname can be set to indicate a FOR BIT DATA string as follows:</td>
</tr>
<tr>
<td></td>
<td>(30)</td>
<td>For columns and parameter markers that have a system-generated name, the thirtieth byte is set to X'FF'. For column names specified by the AS clause, this byte is X'00'.</td>
<td>• The sixth byte of the SQLDAID in the SQLDA header is set to ‘+’</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• The length of sqlname is 8</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• The first two bytes of sqlname are X'0000'</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• The third and fourth bytes of sqlname are X'0000'</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• The remaining four bytes of sqlname are reserved and should be set to X'00000000'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>When working with XML data, sqlname can be set to indicate an XML subtype as follows:</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• The length of sqlname is 8</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• The first two bytes of sqlname are X'0000'</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• The third and fourth bytes of sqlname are X'0000'</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• The remaining three bytes of sqlname are reserved and should be set to X'000000'</td>
<td></td>
</tr>
</tbody>
</table>

**Fields in an occurrence of a secondary SQLVAR**

**Table 74. Fields in a Secondary SQLVAR**

<table>
<thead>
<tr>
<th>Name</th>
<th>Data Type</th>
<th>Usage in DESCRIBE and PREPARE</th>
<th>Usage in FETCH, OPEN, and EXECUTE</th>
</tr>
</thead>
<tbody>
<tr>
<td>len.sqllonglen</td>
<td>INTEGER</td>
<td>The length attribute of a BLOB, CLOB, or DBCLOB column or parameter marker.</td>
<td>The length attribute of a BLOB, CLOB, or DBCLOB host variable. The database manager ignores the SQLLEN field in the Base SQLVAR for the data types. The length attribute stores the number of bytes for a BLOB or CLOB, and the number of double-byte characters for a DBCLOB.</td>
</tr>
<tr>
<td>reserve2</td>
<td>CHAR(3) for 32 bit, and CHAR(11) for 64 bit.</td>
<td>Not used.</td>
<td>Not used.</td>
</tr>
</tbody>
</table>
# SQLDA (SQL descriptor area)

Table 74. Fields in a Secondary SQLVAR (continued)

<table>
<thead>
<tr>
<th>Name</th>
<th>Data Type</th>
<th>Usage in DESCRIBE and PREPARE</th>
<th>Usage in FETCH, OPEN, and EXECUTE</th>
</tr>
</thead>
<tbody>
<tr>
<td>sqlflag4</td>
<td>CHAR(1)</td>
<td>The value is X'01' if the SQLVAR represents a reference type with a target type named in sqldatatype_name. The value is X'12' if the SQLVAR represents a structured type, with the user-defined type name in sqldatatype_name. Otherwise, the value is X'00'.</td>
<td>Set to X'01' if the SQLVAR represents a reference type with a target type named in sqldatatype_name. Set to X'12' if the SQLVAR represents a structured type, with the user-defined type name in sqldatatype_name. Otherwise, the value is X'00'.</td>
</tr>
<tr>
<td>sqldatalen</td>
<td>pointer</td>
<td>Not used.</td>
<td>Used for BLOB, CLOB, and DBCLOB host variables only. If this field is NULL, then the actual length (in double-byte characters) should be stored in the 4 bytes immediately before the start of the data and SQLDATA should point to the first byte of the field length. If this field is not NULL, it contains a pointer to a 4 byte long buffer that contains the actual length in bytes (even for DBCLOB) of the data in the buffer pointed to from the SQLDATA field in the matching base SQLVAR. Note that, whether or not this field is used, the len.sqllonglen field must be set.</td>
</tr>
<tr>
<td>sqldatatype_name</td>
<td>VARCHAR(27)</td>
<td>For a user-defined type, the database manager sets this to the fully qualified user-defined type name. For a reference type, the database manager sets this to the fully qualified type name of the target type of the reference.</td>
<td>For structured types, set to the fully qualified user-defined type name in the format indicated in the table note.</td>
</tr>
<tr>
<td>reserved</td>
<td>CHAR(3)</td>
<td>Not used.</td>
<td>Not used.</td>
</tr>
</tbody>
</table>

1 The first 8 bytes contain the schema name of the type (extended to the right with spaces, if necessary). Byte 9 contains a dot (.). Bytes 10 to 27 contain the low order portion of the type name, which is not extended to the right with spaces.

Note that, although the prime purpose of this field is for the name of user-defined types, the field is also set for IBM predefined data types. In this case, the schema name is SYSIBM, and the low order portion of the name is the name stored in the TYPENAME column of the DATATYPES catalog view. For example:

<table>
<thead>
<tr>
<th>type name</th>
<th>length</th>
<th>sqldatatype_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.B</td>
<td>10</td>
<td>A .B</td>
</tr>
<tr>
<td>INTEGER</td>
<td>16</td>
<td>SYSIBM .INTEGER</td>
</tr>
<tr>
<td>&quot;Frank's&quot;.SMINT</td>
<td>13</td>
<td>Frank's .SMINT</td>
</tr>
<tr>
<td>MY.&quot;type &quot;</td>
<td>15</td>
<td>MY .type</td>
</tr>
</tbody>
</table>
Effect of DESCRIBE on the SQLDA

For a DESCRIBE OUTPUT or PREPARE OUTPUT INTO statement, the database manager always sets SQLD to the number of columns in the result set, or the number of output parameter markers. For a DESCRIBE INPUT or PREPARE INPUT INTO statement, the database manager always sets SQLD to the number of input parameter markers in the statement. Note that a parameter marker that corresponds to an INOUT parameter in a CALL statement is described in both the input and output descriptors.

The SQLVARs in the SQLDA are set in the following cases:

v SQLN >= SQLD and no entry is either a LOB, user-defined type or reference type
   The first SQLD SQLVAR entries are set and SQLDOUBLED is set to blank.

v SQLN >= 2*SQLD and at least one entry is a LOB, user-defined type or reference type
   Two times SQLD SQLVAR entries are set, and SQLDOUBLED is set to ‘2’.

v SQLD <= SQLN < 2*SQLD and at least one entry is a distinct, array, cursor, row, or reference type, but there are no LOB entries or structured type entries
   The first SQLD SQLVAR entries are set and SQLDOUBLED is set to blank. If the SQLWARN bind option is YES, a warning SQLCODE +237 (SQLSTATE 01594) is issued.

The SQLVARs in the SQLDA are NOT set (requiring allocation of additional space and another DESCRIBE) in the following cases:

v SQLN < SQLD and no entry is either a LOB, user-defined type or reference type
   No SQLVAR entries are set and SQLDOUBLED is set to blank. If the SQLWARN bind option is YES, a warning SQLCODE +236 (SQLSTATE 01005) is issued. Allocate SQLD SQLVARs for a successful DESCRIBE.

v SQLN < SQLD and at least one entry is a distinct, array, cursor, row, or reference type, but there are no LOB entries or structured type entries
   No SQLVAR entries are set and SQLDOUBLED is set to blank. If the SQLWARN bind option is YES, a warning SQLCODE +239 (SQLSTATE 01005) is issued. Allocate 2*SQLD SQLVARs for a successful DESCRIBE including the names of the distinct, array, cursor, and row types and target types of reference types.

v SQLN < 2*SQLD and at least one entry is a LOB or a structured type
   No SQLVAR entries are set and SQLDOUBLED is set to blank. A warning SQLCODE +238 (SQLSTATE 01005) is issued (regardless of the setting of the SQLWARN bind option). Allocate 2*SQLD SQLVARs for a successful DESCRIBE.

References in the above lists to LOB entries include distinct type entries whose source type is a LOB type.

The SQLWARN option of the BIND or PREP command is used to control whether the DESCRIBE (or PREPARE INTO) will return the warning SQLCODEs +236, +237, +239. It is recommended that your application code always consider that these SQLCODEs could be returned. The warning SQLCODE +238 is always returned when there are LOB or structured type entries in the select list and there are insufficient SQLVARs in the SQLDA. This is the only way the application can know that the number of SQLVARs must be doubled because of a LOB or structured type entry in the result set.
If a structured type entry is being described, but no FROM SQL transform is defined (either because no TRANSFORM GROUP was specified using the CURRENT DEFAULT TRANSFORM GROUP special register (SQLSTATE 42741) or because the name group does not have a FROM SQL transform function defined (SQLSTATE 42744)), the DESCRIBE will return an error. This error is the same error returned for a DESCRIBE of a table with a structured type entry.

If the database manager returns identifiers that are longer than those that can be stored in the SQLDA, the identifier is truncated and a warning is returned (SQLSTATE 01665); however, when the name of a structured type is truncated, an error is returned (SQLSTATE 42622). For details on identifier length limitations, see “SQL and XQuery limits”.

**SQLTYPE and SQLLEN**

Table 75 shows the values that may appear in the SQLTYPE and SQLLEN fields of the SQLDA. In DESCRIBE and PREPARE INTO, an even value of SQLTYPE means that the column does not allow nulls, and an odd value means the column does allow nulls. In FETCH, OPEN, and EXECUTE, an even value of SQLTYPE means that no indicator variable is provided, and an odd value means that SQLIND contains the address of an indicator variable.

### Table 75: SQLTYPE and SQLLEN values for DESCRIBE, FETCH, OPEN, and EXECUTE

<table>
<thead>
<tr>
<th>SQLTYPE</th>
<th>Column Data Type</th>
<th>SQLLEN</th>
<th>Host Variable Data Type</th>
<th>SQLLEN</th>
</tr>
</thead>
<tbody>
<tr>
<td>384/385</td>
<td>date</td>
<td>10</td>
<td>fixed-length character string representation of a date</td>
<td>length attribute of the host variable</td>
</tr>
<tr>
<td>388/389</td>
<td>time</td>
<td>8</td>
<td>fixed-length character string representation of a time</td>
<td>length attribute of the host variable</td>
</tr>
<tr>
<td>392/393</td>
<td>timestamp</td>
<td>19 for TIMESTAMP(0) otherwise 20+p for TIMESTAMP(p)</td>
<td>fixed-length character string representation of a timestamp</td>
<td>length attribute of the host variable</td>
</tr>
<tr>
<td>400/401</td>
<td>N/A</td>
<td>N/A</td>
<td>NULL-terminated graphic string</td>
<td>length attribute of the host variable</td>
</tr>
<tr>
<td>404/405</td>
<td>BLOB</td>
<td>0 '</td>
<td>BLOB</td>
<td>Not used. *</td>
</tr>
<tr>
<td>408/409</td>
<td>CLOB</td>
<td>0 '</td>
<td>CLOB</td>
<td>Not used. *</td>
</tr>
<tr>
<td>412/413</td>
<td>DBCLOB</td>
<td>0 '</td>
<td>DBCLOB</td>
<td>Not used. *</td>
</tr>
<tr>
<td>448/449</td>
<td>varying-length character string</td>
<td>length attribute of the column</td>
<td>varying-length character string</td>
<td>length attribute of the host variable</td>
</tr>
<tr>
<td>452/453</td>
<td>fixed-length character string</td>
<td>length attribute of the column</td>
<td>fixed-length character string</td>
<td>length attribute of the host variable</td>
</tr>
<tr>
<td>456/457</td>
<td>long varying-length character string</td>
<td>length attribute of the column</td>
<td>long varying-length character string</td>
<td>length attribute of the host variable</td>
</tr>
<tr>
<td>460/461</td>
<td>not applicable</td>
<td>not applicable</td>
<td>NULL-terminated character string</td>
<td>length attribute of the host variable</td>
</tr>
<tr>
<td>464/465</td>
<td>varying-length graphic string</td>
<td>length attribute of the column</td>
<td>varying-length graphic string</td>
<td>length attribute of the host variable</td>
</tr>
<tr>
<td>468/469</td>
<td>fixed-length graphic string</td>
<td>length attribute of the column</td>
<td>fixed-length graphic string</td>
<td>length attribute of the host variable</td>
</tr>
</tbody>
</table>
### SQLDA (SQL descriptor area)

**Table 75. SQLTYPE and SQLLEN values for DESCRIBE, FETCH, OPEN, and EXECUTE (continued)**

<table>
<thead>
<tr>
<th>SQLTYPE</th>
<th>Column Data Type</th>
<th>SQLLEN</th>
<th>Host Variable Data Type</th>
<th>SQLLEN</th>
</tr>
</thead>
<tbody>
<tr>
<td>472/473</td>
<td>long varying-length</td>
<td>length attribute of</td>
<td>long graphic string</td>
<td>length attribute of</td>
</tr>
<tr>
<td></td>
<td>graphic string</td>
<td>the column</td>
<td></td>
<td>the host variable</td>
</tr>
<tr>
<td>480/481</td>
<td>floating-point</td>
<td>8 for double</td>
<td>floating-point</td>
<td>8 for double</td>
</tr>
<tr>
<td></td>
<td></td>
<td>precision, 4 for single</td>
<td></td>
<td>precision, 4 for single</td>
</tr>
<tr>
<td></td>
<td></td>
<td>precision</td>
<td></td>
<td>precision</td>
</tr>
<tr>
<td>484/485</td>
<td>packed decimal</td>
<td>precision in byte 1;</td>
<td>packed decimal</td>
<td>precision in byte 1;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>scale in byte 2</td>
<td></td>
<td>scale in byte 2</td>
</tr>
<tr>
<td>492/493</td>
<td>big integer</td>
<td>8</td>
<td>big integer</td>
<td>8</td>
</tr>
<tr>
<td>496/497</td>
<td>large integer</td>
<td>4</td>
<td>large integer</td>
<td>4</td>
</tr>
<tr>
<td>500/501</td>
<td>small integer</td>
<td>2</td>
<td>small integer</td>
<td>2</td>
</tr>
<tr>
<td>916/917</td>
<td>not applicable</td>
<td>not applicable</td>
<td>BLOB file reference</td>
<td>267</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>variable</td>
<td></td>
</tr>
<tr>
<td>920/921</td>
<td>not applicable</td>
<td>not applicable</td>
<td>CLOB file reference</td>
<td>267</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>variable</td>
<td></td>
</tr>
<tr>
<td>924/925</td>
<td>not applicable</td>
<td>not applicable</td>
<td>DBCLOB file</td>
<td>267</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>reference variable</td>
<td></td>
</tr>
<tr>
<td>960/961</td>
<td>not applicable</td>
<td>not applicable</td>
<td>BLOB locator</td>
<td>4</td>
</tr>
<tr>
<td>964/965</td>
<td>not applicable</td>
<td>not applicable</td>
<td>CLOB locator</td>
<td>4</td>
</tr>
<tr>
<td>968/969</td>
<td>not applicable</td>
<td>not applicable</td>
<td>DBCLOB locator</td>
<td>4</td>
</tr>
<tr>
<td>988/989</td>
<td>XML</td>
<td>0</td>
<td>not applicable; use an</td>
<td>not used</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>XML AS &lt;string or</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>binary LOB type&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>host variable instead</td>
<td></td>
</tr>
<tr>
<td>996</td>
<td>decimal floating-point</td>
<td>8 for DECFLOAT(16),</td>
<td>decimal floating-point</td>
<td>8 for DECFLOAT(16),</td>
</tr>
<tr>
<td></td>
<td></td>
<td>16 for DECFLOAT(34)</td>
<td></td>
<td>16 for DECFLOAT(34)</td>
</tr>
<tr>
<td>2440/2441</td>
<td>row</td>
<td>not applicable</td>
<td>row</td>
<td>not used</td>
</tr>
<tr>
<td>2440/2441</td>
<td>cursor</td>
<td>not applicable</td>
<td>row</td>
<td>not used</td>
</tr>
</tbody>
</table>

**Note:**
- The len.sqllonglen field in the secondary SQLVAR contains the length attribute of the column.
- The SQLTYPE has changed from the previous version for portability in DB2. The values from the previous version (see previous version SQL Reference) continue to be supported.

### Unrecognized and unsupported SQLTYPEs

The values that appear in the SQLTYPE field of the SQLDA are dependent on the level of data type support available at the sender as well as at the receiver of the data. This is particularly important as new data types are added to the product.

New data types may or may not be supported by the sender or receiver of the data and may or may not even be recognized by the sender or receiver of the data. Depending on the situation, the new data type may be returned, or a compatible data type agreed upon by both the sender and receiver of the data may be returned or an error may result.
When the sender and receiver agree to use a compatible data type, the following indicates the mapping that will take place. This mapping will take place when at least one of the sender or the receiver does not support the data type provided. The unsupported data type can be provided by either the application or the database manager.

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Compatible Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIGINT</td>
<td>DECIMAL(19, 0)</td>
</tr>
<tr>
<td>ROWID(^i)</td>
<td>VARCHAR(40) FOR BIT DATA</td>
</tr>
</tbody>
</table>

\(^i\) ROWID is supported by DB2 Universal Database for z/OS Version 8.

Note that no indication is given in the SQLDA that the data type is substituted.

**Packed decimal numbers**

Packed decimal numbers are stored in a variation of Binary Coded Decimal (BCD) notation. In BCD, each nybble (four bits) represents one decimal digit. For example, 0001 0111 1001 represents 179. Therefore, read a packed decimal value nybble by nybble. Store the value in bytes and then read those bytes in hexadecimal representation to return to decimal. For example, 0001 0111 1001 becomes 00000001 01111001 in binary representation. By reading this number as hexadecimal, it becomes 0179.

The decimal point is determined by the scale. In the case of a DEC(12,5) column, for example, the rightmost 5 digits are to the right of the decimal point.

Sign is indicated by a nybble to the right of the nybbles representing the digits. A positive or negative sign is indicated as follows:

<table>
<thead>
<tr>
<th>Sign</th>
<th>Binary</th>
<th>Decimal</th>
<th>Hexadecimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Positive (+)</td>
<td>1100</td>
<td>12</td>
<td>C</td>
</tr>
<tr>
<td>Negative (-)</td>
<td>1101</td>
<td>13</td>
<td>D</td>
</tr>
</tbody>
</table>

In summary:

- To store any value, allocate \(p/2+1\) bytes, where \(p\) is precision.
- Assign the nybbles from left to right to represent the value. If a number has an even precision, a leading zero nybble is added. This assignment includes leading (insignificant) and trailing (significant) zero digits.
- The sign nybble will be the second nybble of the last byte.

For example:

<table>
<thead>
<tr>
<th>Column</th>
<th>Value</th>
<th>Nybbles in Hexadecimal</th>
<th>Grouped by Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEC(8,3)</td>
<td>6574.23</td>
<td>00 65 74 23 0C</td>
<td></td>
</tr>
<tr>
<td>DEC(6,2)</td>
<td>-334.02</td>
<td>00 33 40 2D</td>
<td></td>
</tr>
<tr>
<td>DEC(7,5)</td>
<td>5.2323</td>
<td>05 23 23 0C</td>
<td></td>
</tr>
<tr>
<td>DEC(5,2)</td>
<td>-23.5</td>
<td>02 35 0D</td>
<td></td>
</tr>
</tbody>
</table>
SQLDA (SQL descriptor area)

**SQLLEN field for decimal**

The SQLLEN field contains the precision (first byte) and scale (second byte) of the decimal column. If writing a portable application, the precision and scale bytes should be set individually, versus setting them together as a short integer. This will avoid integer byte reversal problems.

For example, in C:

```c
((char *)(sqlda->sqlvar[i].sqllen))[0] = precision;
((char *)(sqlda->sqlvar[i].sqllen))[1] = scale;
```
Appendix D. System catalog views

The database manager creates and maintains two sets of system catalog views that are defined on top of the base system catalog tables.

- SYSCAT views are read-only catalog views that are found in the SYSCAT schema. The RESTRICT option on CREATE DATABASE statement determines how SELECT privilege is granted. When the RESTRICT option is not specified, SELECT privilege is granted to PUBLIC.

- SYSSTAT views are updatable catalog views that are found in the SYSSTAT schema. The updatable views contain statistical information that is used by the optimizer. The values in some columns in these views can be changed to test performance. (Before changing any statistics, it is recommended that the RUNSTATS command be invoked so that all the statistics reflect the current state.)

Applications should be written to the SYSCAT and SYSSTAT views rather than the base catalog tables.

All the system catalog views are created at database creation time. The catalog views cannot be explicitly created or dropped. In a Unicode database, the catalog views are created with IDENTITY collation. In non-Unicode databases, the catalog views are created with the database collation. The views are updated during normal operation in response to SQL data definition statements, environment routines, and certain utilities. Data in the system catalog views is available through normal SQL query facilities. The system catalog views (with the exception of some updatable catalog views) cannot be modified using normal SQL data manipulation statements.

A object table, column, or index object will appear in a user’s updatable SYSSTAT catalog view only if that user holds CONTROL privilege on the object, or holds explicit DATAACCESS authority. A routine object will appear in a user’s updatable SYSSTAT_ROUTINES catalog view if that user owns the routine or holds SQLADM authority.

The order of columns in the views may change from release to release. To prevent this from affecting programming logic, specify the columns in a select list explicitly, and avoid using SELECT *. Columns have consistent names based on the types of objects that they describe.

Table 77. Samples of consistent column names for objects they describe

<table>
<thead>
<tr>
<th>Described Object</th>
<th>Column Names</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table</td>
<td>TABSCHEMA, TABNAME</td>
</tr>
<tr>
<td>Index</td>
<td>INDSHEMA, INDNAME</td>
</tr>
<tr>
<td>Index extension</td>
<td>IESHEMA, IENAME</td>
</tr>
<tr>
<td>View</td>
<td>VIEWSHEMA, VIEWNAME</td>
</tr>
<tr>
<td>Constraint</td>
<td>CONSTSCHEMA, CONSTNAME</td>
</tr>
<tr>
<td>Trigger</td>
<td>TRIGSCHEMA, TRIGNAME</td>
</tr>
<tr>
<td>Package</td>
<td>PKGSHEMA, PKGNAME</td>
</tr>
<tr>
<td>Type</td>
<td>TYPESHEMA, TYPENAME, TYPEID</td>
</tr>
</tbody>
</table>
### System catalog views

Table 77. Samples of consistent column names for objects they describe (continued)

<table>
<thead>
<tr>
<th>Described Object</th>
<th>Column Names</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function</td>
<td>ROUTINESCHEMA, ROUTINEMODULENAME, ROUTINENAME, ROUTINEID</td>
</tr>
<tr>
<td>Method</td>
<td>ROUTINESCHEMA, ROUTINEMODULENAME, ROUTINENAME, ROUTINEID</td>
</tr>
<tr>
<td>Procedure</td>
<td>ROUTINESCHEMA, ROUTINEMODULENAME, ROUTINENAME, ROUTINEID</td>
</tr>
<tr>
<td>Column</td>
<td>COLNAME</td>
</tr>
<tr>
<td>Schema</td>
<td>SCHEMANAME</td>
</tr>
<tr>
<td>Table Space</td>
<td>TBSPACE</td>
</tr>
<tr>
<td>Database partition group</td>
<td>DBPGNAME</td>
</tr>
<tr>
<td>Audit policy</td>
<td>AUDITPOLICYNAME, AUDITPOLICYID</td>
</tr>
<tr>
<td>Buffer pool</td>
<td>BPNAME</td>
</tr>
<tr>
<td>Event Monitor</td>
<td>EVMONNAME</td>
</tr>
<tr>
<td>Condition</td>
<td>CONDSCHEMA, CONDMODULENAME, CONDNAME, CONDMODULEID</td>
</tr>
<tr>
<td>Data source</td>
<td>SERVERNAME, SERVERTYPE, SERVERVERSION</td>
</tr>
<tr>
<td>Global variable</td>
<td>VARSHEMA, VARMODULENAME, VARNAME, VARMODULEID</td>
</tr>
<tr>
<td>Histogram template</td>
<td>TEMPLATENAME, TEMPLATEID</td>
</tr>
<tr>
<td>Module</td>
<td>MODULESCHEMA, MODULENAME, MODULEID</td>
</tr>
<tr>
<td>Role</td>
<td>ROLENAME, ROLEID</td>
</tr>
<tr>
<td>Security label</td>
<td>SECLABELNAME, SECLABELID</td>
</tr>
<tr>
<td>Security policy</td>
<td>SECPOLICYNAME, SECPOLICYID</td>
</tr>
<tr>
<td>Sequence</td>
<td>SEQSCHEMA, SEQNAME</td>
</tr>
<tr>
<td>Threshold</td>
<td>THRESHOLDNAME, THRESHOLDID</td>
</tr>
<tr>
<td>Trusted context</td>
<td>CONTEXTNAME, CONTEXTID</td>
</tr>
<tr>
<td>Work action</td>
<td>ACTIONNAME, ACTIONID</td>
</tr>
<tr>
<td>Work action set</td>
<td>ACTIONSETNAME, ACTIONSETID</td>
</tr>
<tr>
<td>Work class</td>
<td>WORKCLASSNAME, WORKCLASSID</td>
</tr>
<tr>
<td>Work class set</td>
<td>WORKCLASSSETNAME, WORKCLASSSETID</td>
</tr>
<tr>
<td>Workload</td>
<td>WORKLOADID, WORKLOADNAME</td>
</tr>
<tr>
<td>Wrapper</td>
<td>WRAPNAME</td>
</tr>
<tr>
<td>Alteration</td>
<td>ALTER_TIME</td>
</tr>
<tr>
<td>Timestamp</td>
<td>CREATE_TIME</td>
</tr>
</tbody>
</table>
# Road map to the catalog views

Table 78. Road map to the read-only catalog views

<table>
<thead>
<tr>
<th>Description</th>
<th>Catalog View</th>
</tr>
</thead>
<tbody>
<tr>
<td>attributes of structured data types</td>
<td>“SYSCAT.ATTRIBUTES” on page 778</td>
</tr>
<tr>
<td>audit policies</td>
<td>“SYSCAT.AUDITPOLICIES” on page 780</td>
</tr>
<tr>
<td></td>
<td>“SYSCAT.AUDITUSE” on page 782</td>
</tr>
<tr>
<td>authorities on database</td>
<td>“SYSCAT.DBAUTH” on page 813</td>
</tr>
<tr>
<td>buffer pool configuration on database partition group</td>
<td>“SYSCAT.BUFFERPOOLS” on page 784</td>
</tr>
<tr>
<td>buffer pool size on database partition</td>
<td>“SYSCAT.BUFFERPOOLDBPARTITIONS” on page 783</td>
</tr>
<tr>
<td>cast functions</td>
<td>“SYSCAT.CASTFUNCTIONS” on page 785</td>
</tr>
<tr>
<td>check constraints</td>
<td>“SYSCAT.CHECKS” on page 786</td>
</tr>
<tr>
<td>column privileges</td>
<td>“SYSCAT.COLAUTH” on page 787</td>
</tr>
<tr>
<td>columns</td>
<td>“SYSCAT.COLUMNS” on page 796</td>
</tr>
<tr>
<td>columns referenced by check constraints</td>
<td>“SYSCAT.COLCHECKS” on page 788</td>
</tr>
<tr>
<td>columns used in dimensions</td>
<td>“SYSCAT.COLUSE” on page 801</td>
</tr>
<tr>
<td>columns used in keys</td>
<td>“SYSCAT.KEYCOLUSE” on page 849</td>
</tr>
<tr>
<td>conditions</td>
<td>“SYSCAT.CONDITIONS” on page 802</td>
</tr>
<tr>
<td>constraint dependencies</td>
<td>“SYSCAT.CONSTDEP” on page 803</td>
</tr>
<tr>
<td>database partition group database partitions</td>
<td>“SYSCAT.DBPARTITIONGROUPDEF” on page 815</td>
</tr>
<tr>
<td>database partition group definitions</td>
<td>“SYSCAT.DBPARTITIONGROUPS” on page 816</td>
</tr>
<tr>
<td>data partitions</td>
<td>“SYSCAT.DATAPARTITIONEXPRESSION” on page 806</td>
</tr>
<tr>
<td>data type dependencies</td>
<td>“SYSCAT.DATATYPEDEP” on page 809</td>
</tr>
<tr>
<td>data types</td>
<td>“SYSCAT.DATATYPES” on page 810</td>
</tr>
<tr>
<td>detailed column group statistics</td>
<td>“SYSCAT.COLGROUPCOLS” on page 790</td>
</tr>
<tr>
<td></td>
<td>“SYSCAT.COLGROUPDIST” on page 791</td>
</tr>
<tr>
<td></td>
<td>“SYSCAT.COLGROUPDISTCOUNTS” on page 792</td>
</tr>
<tr>
<td></td>
<td>“SYSCAT.COLGROUPS” on page 793</td>
</tr>
<tr>
<td>detailed column options</td>
<td>“SYSCAT.COLOPTIONS” on page 795</td>
</tr>
<tr>
<td>detailed column statistics</td>
<td>“SYSCAT.COLDIST” on page 789</td>
</tr>
<tr>
<td>distribution maps</td>
<td>“SYSCAT.PARTITIONMAPS” on page 865</td>
</tr>
<tr>
<td>event monitor definitions</td>
<td>“SYSCAT.EVENTMONITORS” on page 817</td>
</tr>
<tr>
<td>events currently monitored</td>
<td>“SYSCAT.EVENTS” on page 819</td>
</tr>
<tr>
<td></td>
<td>“SYSCAT.EVENTTABLES” on page 820</td>
</tr>
<tr>
<td>fields of row data types</td>
<td>“SYSCAT.ROWFIELDS” on page 889</td>
</tr>
<tr>
<td>function dependencies¹</td>
<td>“SYSCAT.ROUTINEDEP” on page 873</td>
</tr>
<tr>
<td>function mapping</td>
<td>“SYSCAT.FUNCMAPPINGS” on page 824</td>
</tr>
<tr>
<td>function mapping options</td>
<td>“SYSCAT.FUNCMAPOPTIONS” on page 822</td>
</tr>
<tr>
<td>function parameter mapping options</td>
<td>“SYSCAT.FUNCMAPPARMOPTIONS” on page 823</td>
</tr>
<tr>
<td>function parameters¹</td>
<td>“SYSCAT.ROUTINEPARMS” on page 877</td>
</tr>
<tr>
<td>functions¹</td>
<td>“SYSCAT.ROUTINES” on page 880</td>
</tr>
</tbody>
</table>
### Road map to the catalog views

Table 78. Road map to the read-only catalog views (continued)

<table>
<thead>
<tr>
<th>Description</th>
<th>Catalog View</th>
</tr>
</thead>
<tbody>
<tr>
<td>global variables</td>
<td>“SYSCAT.VARIABLEAUTH” on page 934</td>
</tr>
<tr>
<td></td>
<td>“SYSCAT.VARIABLEDEP” on page 935</td>
</tr>
<tr>
<td></td>
<td>“SYSCAT.VARIABLES” on page 936</td>
</tr>
<tr>
<td>hierarchies (types, tables, views)</td>
<td>“SYSCAT.HIERARCHIES” on page 825</td>
</tr>
<tr>
<td></td>
<td>“SYSCAT.FULLHIERARCHIES” on page 821</td>
</tr>
<tr>
<td>identity columns</td>
<td>“SYSCAT.COLIDENTATTRIBUTES” on page 794</td>
</tr>
<tr>
<td>index columns</td>
<td>“SYSCAT.INDEXCOLUSE” on page 830</td>
</tr>
<tr>
<td>index data partitions</td>
<td>“SYSCAT.INDEXPARTITIONS” on page 844</td>
</tr>
<tr>
<td>index dependencies</td>
<td>“SYSCAT.INDEXDEP” on page 831</td>
</tr>
<tr>
<td>index exploitation</td>
<td>“SYSCAT.INDEXEXPLOITRULES” on page 838</td>
</tr>
<tr>
<td>index extension dependencies</td>
<td>“SYSCAT.INDEXEXTENSIONDEP” on page 839</td>
</tr>
<tr>
<td>index extension parameters</td>
<td>“SYSCAT.INDEXEXTENSIONPARMS” on page 841</td>
</tr>
<tr>
<td>index extension search methods</td>
<td>“SYSCAT.INDEXEXTENSIONMETHODS” on page 840</td>
</tr>
<tr>
<td>index extensions</td>
<td>“SYSCAT.INDEXEXTENSIONS” on page 842</td>
</tr>
<tr>
<td>index options</td>
<td>“SYSCAT.INDEXOPTIONS” on page 843</td>
</tr>
<tr>
<td>index privileges</td>
<td>“SYSCAT.INDEXAUTH” on page 829</td>
</tr>
<tr>
<td>indexes</td>
<td>“SYSCAT.INDEXES” on page 832</td>
</tr>
<tr>
<td>invalid objects</td>
<td>“SYSCAT.INVALIDOBJECTS” on page 848</td>
</tr>
<tr>
<td>method dependencies</td>
<td>“SYSCAT.ROUTINEDEP” on page 873</td>
</tr>
<tr>
<td>method parameters</td>
<td>“SYSCAT.ROUTINES” on page 880</td>
</tr>
<tr>
<td>methods</td>
<td>“SYSCAT.ROUTINES” on page 880</td>
</tr>
<tr>
<td>module objects</td>
<td>“SYSCAT.MODULEOBJECTS” on page 851</td>
</tr>
<tr>
<td>module privileges</td>
<td>“SYSCAT.MODULEAUTH” on page 850</td>
</tr>
<tr>
<td>modules</td>
<td>“SYSCAT.MODULES” on page 852</td>
</tr>
<tr>
<td>nicknames</td>
<td>“SYSCAT.NICKNAMES” on page 854</td>
</tr>
<tr>
<td>object mapping</td>
<td>“SYSCAT.NAMEMAPPINGS” on page 855</td>
</tr>
<tr>
<td>package dependencies</td>
<td>“SYSCAT.PACKAGEDEP” on page 858</td>
</tr>
<tr>
<td>package privileges</td>
<td>“SYSCAT.PACKAGEAUTH” on page 857</td>
</tr>
<tr>
<td>packages</td>
<td>“SYSCAT.PACKAGES” on page 860</td>
</tr>
<tr>
<td>partitioned tables</td>
<td>“SYSCAT.TABDETACHEDDEP” on page 913</td>
</tr>
<tr>
<td>pass-through privileges</td>
<td>“SYSCAT.PASSTHRUAUTH” on page 866</td>
</tr>
<tr>
<td>predicate specifications</td>
<td>“SYSCAT.PREDICATESPECS” on page 867</td>
</tr>
<tr>
<td>procedure options</td>
<td>“SYSCAT ROUTINEOPTIONS” on page 875</td>
</tr>
<tr>
<td>procedure parameter options</td>
<td>“SYSCAT ROUTINEPARM OPTIONS” on page 876</td>
</tr>
<tr>
<td>procedure parameters</td>
<td>“SYSCAT ROUTINEPARMS” on page 877</td>
</tr>
<tr>
<td>procedures</td>
<td>“SYSCAT ROUTINES” on page 880</td>
</tr>
</tbody>
</table>
### Table 78. Road map to the read-only catalog views (continued)

<table>
<thead>
<tr>
<th>Description</th>
<th>Catalog View</th>
</tr>
</thead>
<tbody>
<tr>
<td>protected tables</td>
<td>“SYSCAT.SECURITYLABELACCESS” on page 892</td>
</tr>
<tr>
<td></td>
<td>“SYSCAT.SECURITYLABELCOMPONENTELEMENTS” on page 893</td>
</tr>
<tr>
<td></td>
<td>“SYSCAT.SECURITYLABELCOMPONENTS” on page 894</td>
</tr>
<tr>
<td></td>
<td>“SYSCAT.SECURITYLABELS” on page 895</td>
</tr>
<tr>
<td></td>
<td>“SYSCAT.SECURITYPOLICIES” on page 896</td>
</tr>
<tr>
<td></td>
<td>“SYSCAT.SECURITYPOLICYCOMPONENTRULES” on page 897</td>
</tr>
<tr>
<td></td>
<td>“SYSCAT.SECURITYPOLICYEXEMPTIONS” on page 898</td>
</tr>
<tr>
<td></td>
<td>“SYSCAT.SURROGATEAUTHIDS” on page 907</td>
</tr>
<tr>
<td>provides DB2 for z/OS compatibility</td>
<td>“SYSIBM.SYSDUMMY1” on page 961</td>
</tr>
<tr>
<td>referential constraints</td>
<td>“SYSCAT.REFERENCES” on page 868</td>
</tr>
<tr>
<td>remote table options</td>
<td>“SYSCAT.TABOPTIONS” on page 922</td>
</tr>
<tr>
<td>roles</td>
<td>“SYSCAT.ROLEAUTH” on page 869</td>
</tr>
<tr>
<td></td>
<td>“SYSCAT.ROLES” on page 870</td>
</tr>
<tr>
<td>routine dependencies</td>
<td>“SYSCAT_ROUTINEDEP” on page 873</td>
</tr>
<tr>
<td>routine parameters(^1)</td>
<td>“SYSCAT_ROUTINEPARMS” on page 877</td>
</tr>
<tr>
<td>routine privileges</td>
<td>“SYSCAT_ROUTINEAUTH” on page 871</td>
</tr>
<tr>
<td>routines(^1)</td>
<td>“SYSCAT_ROUTINES” on page 880</td>
</tr>
<tr>
<td></td>
<td>“SYSCAT_ROUTINESFEDERATED” on page 887</td>
</tr>
<tr>
<td>schema privileges</td>
<td>“SYSCAT_SCHEMAAUTH” on page 890</td>
</tr>
<tr>
<td>schemas</td>
<td>“SYSCAT_SCHEMA” on page 891</td>
</tr>
<tr>
<td>sequence privileges</td>
<td>“SYSCAT_SEQUENCEAUTH” on page 899</td>
</tr>
<tr>
<td>sequences</td>
<td>“SYSCAT_SEQUENCES” on page 900</td>
</tr>
<tr>
<td>server options</td>
<td>“SYSCAT_SERVEROPTIONS” on page 902</td>
</tr>
<tr>
<td>server-specific user options</td>
<td>“SYSCAT_USEROPTIONS” on page 933</td>
</tr>
<tr>
<td>statements in packages</td>
<td>“SYSCAT_STATEMENTS” on page 906</td>
</tr>
<tr>
<td>stored procedures</td>
<td>“SYSCAT_ROUTINES” on page 880</td>
</tr>
<tr>
<td>system servers</td>
<td>“SYSCAT_SERVERS” on page 903</td>
</tr>
<tr>
<td>table constraints</td>
<td>“SYSCAT_TABCONST” on page 910</td>
</tr>
<tr>
<td>table dependencies</td>
<td>“SYSCAT_TABDEP” on page 911</td>
</tr>
<tr>
<td>table privileges</td>
<td>“SYSCAT_TABAUTH” on page 908</td>
</tr>
<tr>
<td>table space use privileges</td>
<td>“SYSCAT_TBSPACEAUTH” on page 923</td>
</tr>
<tr>
<td>table spaces</td>
<td>“SYSCAT_TABLESPACES” on page 920</td>
</tr>
<tr>
<td>tables</td>
<td>“SYSCAT_TABLES” on page 914</td>
</tr>
<tr>
<td>transforms</td>
<td>“SYSCAT_TRANSFORMS” on page 926</td>
</tr>
<tr>
<td>trigger dependencies</td>
<td>“SYSCAT_TRIGDEP” on page 927</td>
</tr>
<tr>
<td>triggers</td>
<td>“SYSCAT_TRIGGERS” on page 928</td>
</tr>
<tr>
<td>trusted contexts</td>
<td>“SYSCAT_CONTEXTATTRIBUTES” on page 804</td>
</tr>
<tr>
<td></td>
<td>“SYSCATCONTEXTS” on page 805</td>
</tr>
</tbody>
</table>
### Road map to the catalog views

Table 78. Road map to the read-only catalog views (continued)

<table>
<thead>
<tr>
<th>Description</th>
<th>Catalog View</th>
</tr>
</thead>
<tbody>
<tr>
<td>type mapping</td>
<td>&quot;SYSCAT.TYPEMAPPINGS&quot; on page 930</td>
</tr>
<tr>
<td>user-defined functions</td>
<td>&quot;SYSCAT.ROUTINES&quot; on page 880</td>
</tr>
<tr>
<td>view dependencies</td>
<td>&quot;SYSCAT.TABDEP&quot; on page 911</td>
</tr>
<tr>
<td>views</td>
<td>&quot;SYSCAT.TABLES&quot; on page 914</td>
</tr>
<tr>
<td></td>
<td>&quot;SYSCAT.VIEWS&quot; on page 938</td>
</tr>
<tr>
<td>workload management</td>
<td>&quot;SYSCAT.HISTOGRAMTEMPLATEBINS&quot; on page 826</td>
</tr>
<tr>
<td></td>
<td>&quot;SYSCAT.HISTOGRAMTEMPLATES&quot; on page 827</td>
</tr>
<tr>
<td></td>
<td>&quot;SYSCAT.HISTOGRAMTEMPLATEUSE&quot; on page 828</td>
</tr>
<tr>
<td></td>
<td>&quot;SYSCAT.SERVICECLASSES&quot; on page 904</td>
</tr>
<tr>
<td></td>
<td>&quot;SYSCAT.THRESHOLDS&quot; on page 924</td>
</tr>
<tr>
<td></td>
<td>&quot;SYSCAT.WORKACTIONS&quot; on page 939</td>
</tr>
<tr>
<td></td>
<td>&quot;SYSCAT.WORKACTIONSETS&quot; on page 942</td>
</tr>
<tr>
<td></td>
<td>&quot;SYSCAT.WORKCLASSES&quot; on page 943</td>
</tr>
<tr>
<td></td>
<td>&quot;SYSCAT.WORKCLASSSETS&quot; on page 944</td>
</tr>
<tr>
<td></td>
<td>&quot;SYSCAT.WORKLOADAUTH&quot; on page 945</td>
</tr>
<tr>
<td></td>
<td>&quot;SYSCAT.WORKLOADCONNATTR&quot; on page 946</td>
</tr>
<tr>
<td></td>
<td>&quot;SYSCAT.WORKLOADS&quot; on page 947</td>
</tr>
<tr>
<td>wrapper options</td>
<td>&quot;SYSCAT.WRAPOPTIONS&quot; on page 949</td>
</tr>
<tr>
<td>wrappers</td>
<td>&quot;SYSCAT.WRAPPERS&quot; on page 950</td>
</tr>
<tr>
<td>XML strings</td>
<td>&quot;SYSCAT.XMLSTRINGS&quot; on page 953</td>
</tr>
<tr>
<td>XML values index</td>
<td>&quot;SYSCAT.INDEXXMLPATTERNS&quot; on page 847</td>
</tr>
<tr>
<td>XSR objects</td>
<td>&quot;SYSCAT.XDBMAPGRAPHS&quot; on page 951</td>
</tr>
<tr>
<td></td>
<td>&quot;SYSCAT.XDBMAPSHREDTREES&quot; on page 952</td>
</tr>
<tr>
<td></td>
<td>&quot;SYSCAT.XSROBJECTAUTH&quot; on page 954</td>
</tr>
<tr>
<td></td>
<td>&quot;SYSCAT.XSROBJECTCOMPONENTS&quot; on page 955</td>
</tr>
<tr>
<td></td>
<td>&quot;SYSCAT.XSROBJECTDEP&quot; on page 956</td>
</tr>
<tr>
<td></td>
<td>&quot;SYSCAT.XSROBJECTDETAILS&quot; on page 958</td>
</tr>
<tr>
<td></td>
<td>&quot;SYSCAT.XSROBJECTHIERARCHIES&quot; on page 959</td>
</tr>
<tr>
<td></td>
<td>&quot;SYSCAT.XSROBJECTS&quot; on page 960</td>
</tr>
</tbody>
</table>

1 The following catalog views for functions, methods, and procedures defined in DB2 Version 7.1 and earlier are still available:

- **Functions**: SYSCAT.FUNCTIONS, SYSCAT.FUNCDEP, SYSCAT.FUNCPARAMS
- **Methods**: SYSCAT.FUNCTIONS, SYSCAT.FUNCDEP, SYSCAT.FUNCPARAMS
- **Procedures**: SYSCAT.PROCEDURES, SYSCAT.PROCPARAMS

However, these views have not been updated since DB2 Version 7.1. Use the SYSCAT.ROUTINES, SYSCAT.ROUTINEDEPE, or SYSCAT.ROUTINEPARMS catalog view instead.

Table 79. Road map to the updatable catalog views

<table>
<thead>
<tr>
<th>Description</th>
<th>Catalog View</th>
</tr>
</thead>
<tbody>
<tr>
<td>columns</td>
<td>&quot;SYSSSTAT.COLUMNS&quot; on page 966</td>
</tr>
</tbody>
</table>
Table 79. Road map to the updatable catalog views (continued)

<table>
<thead>
<tr>
<th>Description</th>
<th>Catalog View</th>
</tr>
</thead>
<tbody>
<tr>
<td>detailed column group statistics</td>
<td>“SYSSTAT.COLGROUPDIST” on page 963</td>
</tr>
<tr>
<td></td>
<td>“SYSSTAT.COLGROUPDISTCOUNTS” on page 964</td>
</tr>
<tr>
<td></td>
<td>“SYSSTAT.COLGROUPS” on page 965</td>
</tr>
<tr>
<td>detailed column statistics</td>
<td>“SYSSTAT.COLDIST” on page 962</td>
</tr>
<tr>
<td>indexes</td>
<td>“SYSSTAT.INDEXES” on page 968</td>
</tr>
<tr>
<td>routines(^1)</td>
<td>“SYSSTAT.ROUTINES” on page 972</td>
</tr>
<tr>
<td>tables</td>
<td>“SYSSTAT.TABLES” on page 973</td>
</tr>
</tbody>
</table>

\(^1\) The SYSSTAT.FUNCTIONS catalog view still exists for updating the statistics for functions and methods. This view, however, does not reflect any changes since DB2 Version 7.1.
Each row represents an attribute that is defined for a user-defined structured data type. Includes inherited attributes of subtypes.

### Table 80. SYSCAT.ATTRIBUTES Catalog View

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TYPESCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the structured data type that includes the attribute.</td>
</tr>
<tr>
<td>TYPE_MODULE_NAME</td>
<td>VARCHAR(128) Y</td>
<td></td>
<td>Unqualified name of the module to which the structured data type belongs.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Y</td>
<td>The null value if not a module structured data type.</td>
</tr>
<tr>
<td>TYPENAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the structured data type that includes the attribute.</td>
</tr>
<tr>
<td>ATTR_NAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Attribute name.</td>
</tr>
<tr>
<td>ATTR_TYPESCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the data type of an attribute.</td>
</tr>
<tr>
<td>ATTR_TYPE_MODULE_NAME</td>
<td>VARCHAR(128) Y</td>
<td></td>
<td>Unqualified name of the module to which the data type of an attribute belongs. The null value if not a module attribute.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>ATTR_TYPENAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the data type of an attribute.</td>
</tr>
<tr>
<td>TARGET_TYPESCHEMA</td>
<td>VARCHAR (128) Y</td>
<td></td>
<td>Schema name of the target row type. Applies to reference types only; null value otherwise.</td>
</tr>
<tr>
<td>TARGET_TYPE_MODULE_NAME</td>
<td>VARCHAR(128) Y</td>
<td></td>
<td>Unqualified name of the module to which the target row type belongs. The null value if not a module row type. Applies to reference types only; null value otherwise.</td>
</tr>
<tr>
<td>TARGET_TYPENAME</td>
<td>VARCHAR (128) Y</td>
<td></td>
<td>Unqualified name of the target row type. Applies to reference types only; null value otherwise.</td>
</tr>
<tr>
<td>SOURCE_TYPESCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>For inherited attributes, the schema name of the data type with which the attribute was first defined. For non-inherited attributes, this column is the same as TYPESCHEMA.</td>
</tr>
<tr>
<td>SOURCE_TYPE_MODULE_NAME</td>
<td>VARCHAR(128) Y</td>
<td></td>
<td>For inherited attributes, the unqualified name of the module to which the data type with which the attribute was first defined belongs. For non-inherited attributes, this column is the same as MODULEID. The null value if not a module data type.</td>
</tr>
<tr>
<td>SOURCE_TYPENAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>For inherited attributes, the unqualified name of the data type with which the attribute was first defined. For non-inherited attributes, this column is the same as TYPENAME.</td>
</tr>
<tr>
<td>ORDINAL</td>
<td>SMALLINT</td>
<td></td>
<td>Position of the attribute in the definition of the structured data type, starting with 0.</td>
</tr>
<tr>
<td>LENGTH</td>
<td>INTEGER</td>
<td></td>
<td>Length of the attribute data type. 0 if the attribute is a user-defined type.</td>
</tr>
</tbody>
</table>
### Table 80. SYSCAT.ATTRIBUTES Catalog View (continued)

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCALE</td>
<td>SMALLINT</td>
<td></td>
<td>Scale if the attribute data type is DECIMAL or distinct type based on DECIMAL; the number of digits of fractional seconds if the attribute data type is TIMESTAMP or distinct type based on TIMESTAMP; 0 otherwise.</td>
</tr>
<tr>
<td>CODEPAGE</td>
<td>SMALLINT</td>
<td></td>
<td>For string types, denotes the code page; 0 indicates FOR BIT DATA; 0 for non-string types.</td>
</tr>
<tr>
<td>COLLATIONSCHEMA</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>For string types, the schema name of the collation for the attribute; the null value otherwise.</td>
</tr>
<tr>
<td>COLLATIONNAME</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>For string types, the unqualified name of the collation for the attribute; the null value otherwise.</td>
</tr>
<tr>
<td>LOGGED</td>
<td>CHAR (1)</td>
<td></td>
<td>Applies to LOB types only; blank otherwise.</td>
</tr>
<tr>
<td>COMPACT</td>
<td>CHAR (1)</td>
<td></td>
<td>Applies to LOB types only; blank otherwise.</td>
</tr>
<tr>
<td>DL_FEATURES</td>
<td>CHAR (10)</td>
<td></td>
<td>This column is no longer used and will be removed in a future release.</td>
</tr>
<tr>
<td>JAVA_FIELDNAME</td>
<td>VARCHAR (256)</td>
<td>Y</td>
<td>Reserved for future use.</td>
</tr>
</tbody>
</table>
Table 81. SYSCAT.AUDITPOLICIES Catalog View

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AUDITPOLICYNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Name of the audit policy.</td>
</tr>
<tr>
<td>AUDITPOLICYID</td>
<td>INTEGER</td>
<td></td>
<td>Identifier for the audit policy.</td>
</tr>
<tr>
<td>CREATE_TIME</td>
<td>TIMESTAMP</td>
<td></td>
<td>Time at which the audit policy was created.</td>
</tr>
<tr>
<td>ALTER_TIME</td>
<td>TIMESTAMP</td>
<td></td>
<td>Time at which the audit policy was last altered.</td>
</tr>
<tr>
<td>AUDITSTATUS</td>
<td>CHAR (1)</td>
<td></td>
<td>Status for the AUDIT category.</td>
</tr>
<tr>
<td>CONTEXTSTATUS</td>
<td>CHAR (1)</td>
<td></td>
<td>Status for the CONTEXT category.</td>
</tr>
<tr>
<td>VALIDATESTATUS</td>
<td>CHAR (1)</td>
<td></td>
<td>Status for the VALIDATE category.</td>
</tr>
<tr>
<td>CHECKINGSTATUS</td>
<td>CHAR (1)</td>
<td></td>
<td>Status for the CHECKING category.</td>
</tr>
<tr>
<td>SECMAINTSTATUS</td>
<td>CHAR (1)</td>
<td></td>
<td>Status for the SECMAINT category.</td>
</tr>
<tr>
<td>OBJMAINTSTATUS</td>
<td>CHAR (1)</td>
<td></td>
<td>Status for the OBJMAINT category.</td>
</tr>
<tr>
<td>SYSADMINSTATUS</td>
<td>CHAR (1)</td>
<td></td>
<td>Status for the SYSADMIN category.</td>
</tr>
</tbody>
</table>

Each row represents an audit policy.
### Table 81. SYSCAT.AUDITPOLICIES Catalog View (continued)

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXECUTESTATUS</td>
<td>CHAR (1)</td>
<td></td>
<td>Status for the EXECUTE category.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• B = Both</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• F = Failure</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = None</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• S = Success</td>
</tr>
<tr>
<td>EXECUTEWITHDATA</td>
<td>CHAR (1)</td>
<td></td>
<td>Host variables and parameter markers logged with EXECUTE category.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = No</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = Yes</td>
</tr>
<tr>
<td>ERRORTYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>The audit error type.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• A = Audit</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = Normal</td>
</tr>
<tr>
<td>REMARKS</td>
<td>VARCHAR (254)</td>
<td>Y</td>
<td>User-provided comments, or the null value.</td>
</tr>
</tbody>
</table>
Each row represents an audit policy that is associated with a non-database object, such as USER, GROUP, or authority (SYSADM, SYSCtrl, SYSMAINT).

Table 82. SYSCAT.AUDITUSE Catalog View

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AUDITPOLICYNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Name of the audit policy.</td>
</tr>
<tr>
<td>AUDITPOLICYID</td>
<td>INTEGER</td>
<td></td>
<td>Identifier for the audit policy.</td>
</tr>
<tr>
<td>OBJECTTYPE</td>
<td>CHAR(1)</td>
<td></td>
<td>The type of object with which this audit policy is associated.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- ( S = MQT )</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- ( T = Table )</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- ( g = Authority )</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- ( i = Authorization ID )</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- ( x = Trusted context )</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Blank = Database</td>
</tr>
<tr>
<td>SUBOBJECTTYPE</td>
<td>CHAR(1)</td>
<td></td>
<td>If OBJECTTYPE is ‘i’, this is the type that the authorization ID represents.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- ( G = Group )</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- ( R = Role )</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- ( U = User )</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Blank = Not applicable</td>
</tr>
<tr>
<td>OBJECTSCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the object for which the audit policy is in use.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>OBJECTSCHEMA is null if OBJECTTYPE identifies an object to which a schema does not apply.</td>
</tr>
<tr>
<td>OBJECTNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the object for which this audit policy is in use.</td>
</tr>
</tbody>
</table>
Each row represents a combination of a buffer pool and a database partition, in which the size of the buffer pool on that partition is different from its default size for other partitions in the same database partition group (as represented in SYSCAT.BUFFERPOOLS).

Table 83. SYSCAT.BUFFERPOOLDBPARTITIONS Catalog View

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BUFFERPOOLID</td>
<td>INTEGER</td>
<td></td>
<td>Internal buffer pool identifier.</td>
</tr>
<tr>
<td>DBPARTITIONNUM</td>
<td>SMALLINT</td>
<td></td>
<td>Database partition number.</td>
</tr>
<tr>
<td>NPAGES</td>
<td>INTEGER</td>
<td></td>
<td>Number of pages in this buffer pool on this database partition.</td>
</tr>
</tbody>
</table>
SYSCAT.BUFFERPOOLS

Each row represents the configuration of a buffer pool on one database partition group of a database, or on all database partitions of a database.

Table 84. SYSCAT.BUFFERPOOLS Catalog View

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BPNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Name of the buffer pool.</td>
</tr>
<tr>
<td>BUFFERPOOLID</td>
<td>INTEGER</td>
<td></td>
<td>Identifier for the buffer pool.</td>
</tr>
<tr>
<td>DBPGNAME</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>Name of the database partition group (the null value if the buffer pool exists on all database partitions in the database).</td>
</tr>
<tr>
<td>NPAGES</td>
<td>INTEGER</td>
<td></td>
<td>Default number of pages in this buffer pool on database partitions in this database partition group.</td>
</tr>
<tr>
<td>PAGESIZE</td>
<td>INTEGER</td>
<td></td>
<td>Page size for this buffer pool on database partitions in this database partition group.</td>
</tr>
<tr>
<td>ESTORE</td>
<td>INTEGER</td>
<td></td>
<td>Always ‘N’. Extended storage no longer applies.</td>
</tr>
<tr>
<td>NUMBLOCKPAGES</td>
<td>INTEGER</td>
<td></td>
<td>Number of pages of the buffer pool that are to be in a block-based area. A block-based area of the buffer pool is only used by prefetchers doing a sequential prefetch.</td>
</tr>
<tr>
<td>BLOCKSIZE</td>
<td>INTEGER</td>
<td></td>
<td>Number of pages in a block.</td>
</tr>
<tr>
<td>NGNAME(^1)</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>Name of the database partition group (the null value if the buffer pool exists on all database partitions in the database).</td>
</tr>
</tbody>
</table>

Note:

1. The NGNAME column is included for backwards compatibility. See DBPGNAME.
Each row represents a cast function, not including built-in cast functions.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FROM_TYPESCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the data type of the parameter.</td>
</tr>
<tr>
<td>FROM_TYPENAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Name of the data type of the parameter.</td>
</tr>
<tr>
<td>FROM_TYPEMODULEID</td>
<td>INTEGER</td>
<td>Y</td>
<td>Identifier for the module to which the data type of the parameter belongs.</td>
</tr>
<tr>
<td>TO_TYPESCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the data type of the result after casting.</td>
</tr>
<tr>
<td>TO_TYPENAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Name of the data type of the result after casting.</td>
</tr>
<tr>
<td>TO_TYPEMODULEID</td>
<td>INTEGER</td>
<td>Y</td>
<td>Identifier for the module to which the data type of the result after casting</td>
</tr>
<tr>
<td>FUNCSCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the function.</td>
</tr>
<tr>
<td>FUNCNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the function.</td>
</tr>
<tr>
<td>SPECIFICNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Name of the routine instance (might be system-generated).</td>
</tr>
<tr>
<td>FUNCMODULEID</td>
<td>INTEGER</td>
<td>Y</td>
<td>Identifier for the module to which the function belongs.</td>
</tr>
<tr>
<td>ASSIGN_FUNCTION</td>
<td>CHAR (1)</td>
<td></td>
<td>• N = Not an assignment function</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = Implicit assignment function</td>
</tr>
</tbody>
</table>
SYSCAT.CHECKS

Each row represents a check constraint or a derived column in a materialized query table. For table hierarchies, each check constraint is recorded only at the level of the hierarchy where the constraint was created.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONSTNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Name of the check constraint.</td>
</tr>
<tr>
<td>OWNER</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Authorization ID of the owner of the constraint.</td>
</tr>
</tbody>
</table>
| OWNERTYPE         | CHAR (1) |          | • S = The owner is the system  
                     • U = The owner is an individual user                                       |
| TABSCHEMA         | VARCHAR (128) |          | Schema name of the table to which this constraint applies.                  |
| TABNAME           | VARCHAR (128) |          | Name of the table to which this constraint applies.                         |
| CREATE_TIME       | TIMESTAMP |          | Time at which the constraint was defined. Used in resolving functions that are part of this constraint. Functions that were created after the constraint was defined are not chosen. |
| QUALIFIER         | VARCHAR (128) |          | Value of the default schema at the time of object definition. Used to complete any unqualified references. |
| TYPE              | CHAR (1) |          | Type of check constraint:  
                     • C = Check constraint  
                     • F = Functional dependency  
                     • O = Constraint is an object property  
                     • S = System-generated check constraint for a GENERATED ALWAYS column |
| FUNC_PATH         | CLOB (2K) |          | SQL path in effect when the constraint was defined.                         |
| TEXT              | CLOB (2M) |          | Text of the check condition or definition of the derived column.¹          |
| PERCENTVALID      | SMALLINT |          | Number of rows for which the informational constraint is valid, expressed as a percentage of the total. |
| COLLATIONSHEMA    | VARCHAR (128) |          | Schema name of the collation for the constraint.                           |
| COLLATIONNAME     | VARCHAR (128) |          | Unqualified name of the collation for the constraint.                      |
| COLLATIONSHEMA_ORDERBY | VARCHAR (128) |          | Schema name of the collation for ORDER BY clauses in the constraint.     |
| COLLATIONNAME_ORDERBY | VARCHAR (128) |          | Unqualified name of the collation for ORDER BY clauses in the constraint. |
| DEFINER²         | VARCHAR (128) |          | Authorization ID of the owner of the constraint.                           |

Note:
1. In the catalog view, the text of the check condition is always shown in the database code page and can contain substitution characters. The check constraint will always be applied in the code page of the target table, and will not contain any substitution characters when applied. (The check constraint will be applied based on the original text in the code page of the target table, which might not include the substitution characters.)
2. The DEFINER column is included for backwards compatibility. See OWNER.
Each row represents a user, group, or role that has been granted one or more privileges on a column.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRANTOR</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Grantor of a privilege.</td>
</tr>
<tr>
<td>GRANTORTYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>• S = Grantor is the system</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• U = Grantor is an individual user</td>
</tr>
<tr>
<td>GRANTEE</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Holder of a privilege.</td>
</tr>
<tr>
<td>GRANTEETYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>• G = Grantee is a group</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• R = Grantee is a role</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• U = Grantee is an individual user</td>
</tr>
<tr>
<td>TABSCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the table or view on which the</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>privilege is held.</td>
</tr>
<tr>
<td>TABNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the table or view on which</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>the privilege is held.</td>
</tr>
<tr>
<td>COLNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Name of the column to which this privilege</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>applies.</td>
</tr>
<tr>
<td>COLNO</td>
<td>SMALLINT</td>
<td></td>
<td>Column number of this column within the table</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(starting with 0).</td>
</tr>
<tr>
<td>PRIVTYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>• R = Reference privilege</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• U = Update privilege</td>
</tr>
<tr>
<td>GRANTABLE</td>
<td>CHAR (1)</td>
<td></td>
<td>• G = Privilege is grantable</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = Privilege is not grantable</td>
</tr>
</tbody>
</table>

**Note:**

1. Privileges can be granted by column, but can be revoked only on a table-wide basis.
Each row represents a column that is referenced by a check constraint or by the definition of a materialized query table. For table hierarchies, each check constraint is recorded only at the level of the hierarchy where the constraint was created.

Table 88. SYSCAT.COLCHECKS Catalog View

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONSTNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Name of the check constraint.</td>
</tr>
<tr>
<td>TABSCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the table containing the referenced column.</td>
</tr>
<tr>
<td>TABNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the table containing the referenced column.</td>
</tr>
<tr>
<td>COLNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Name of the column.</td>
</tr>
<tr>
<td>USAGE</td>
<td>CHAR (1)</td>
<td></td>
<td>• D = Column is the child in a functional dependency</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• P = Column is the parent in a functional dependency</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• R = Column is referenced in the check constraint</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• S = Column is a source in the system-generated column check constraint</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• T = Column is a target in the system-generated column check constraint</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>that supports a materialized query table</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>that supports a materialized query table</td>
</tr>
</tbody>
</table>
Each row represents the \( n \)th most frequent value of some column, or the \( n \)th quantile (cumulative distribution) value of the column. Applies to columns of real tables only (not views). No statistics are recorded for inherited columns of typed tables.

Table 89. SYSCAT.COLDIST Catalog View

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TABSCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the table to which the statistics apply.</td>
</tr>
<tr>
<td>TABNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the table to which the statistics apply.</td>
</tr>
<tr>
<td>COLNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Name of the column to which the statistics apply.</td>
</tr>
</tbody>
</table>
| TYPE        | CHAR (1) |          | • \( F \) = Frequency value  
• \( Q \) = Quantile value |
| SEQNO       | SMALLINT |          | If TYPE = ‘F’, \( n \) in this column identifies the \( n \)th most frequent value. If TYPE = ‘Q’, \( n \) in this column identifies the \( n \)th quantile value. |
| COLVALUE\(^1\) | VARCHAR (254) | Y | Data value as a character literal or a null value. |
| VALCOUNT    | BIGINT |          | If TYPE = ‘F’, VALCOUNT is the number of occurrences of COLVALUE in the column. If TYPE = ‘Q’, VALCOUNT is the number of rows whose value is less than or equal to COLVALUE. |
| DISTCOUNT\(^2\) | BIGINT | Y | If TYPE = ‘Q’, this column records the number of distinct values that are less than or equal to COLVALUE (the null value if unavailable). |

Note:

1. In the catalog view, the value of COLVALUE is always shown in the database code page and can contain substitution characters. However, the statistics are gathered internally in the code page of the column’s table, and will therefore use actual column values when applied during query optimization.

2. DISTCOUNT is collected only for columns that are the first key column in an index.
SYSCAT.COLGROUPCOLS

Each row represents a column that makes up a column group.

Table 90. SYSCAT.COLGROUPCOLS Catalog View

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>COLGROUPID</td>
<td>INTEGER</td>
<td></td>
<td>Identifier for the column group.</td>
</tr>
<tr>
<td>COLNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Name of the column in the column group.</td>
</tr>
<tr>
<td>TABSCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the table for the column in the column group.</td>
</tr>
<tr>
<td>TABNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the table for the column in the column group.</td>
</tr>
<tr>
<td>ORDINAL</td>
<td>SMALLINT</td>
<td></td>
<td>Ordinal number of the column in the column group.</td>
</tr>
</tbody>
</table>
Each row represents the value of the column in a column group that makes up the $n$th most frequent value of the column group or the $n$th quantile value of the column group.

### Table 91. SYSCAT.COLGROUPDIST Catalog View

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>COLGROUPID</td>
<td>INTEGER</td>
<td></td>
<td>Identifier for the column group.</td>
</tr>
</tbody>
</table>
| TYPE        | CHAR (1)  |          | • F = Frequency value  
            |            | • Q = Quantile value          |
| ORDINAL     | SMALLINT  |          | Ordinal number of the column in the column group. |
| SEQNO       | SMALLINT  |          | If TYPE = ‘F’, $n$ in this column identifies the $n$th most frequent value. If TYPE = ‘Q’, $n$ in this column identifies the $n$th quantile value. |
| COLVALUE$^1$| VARCHAR (254) |          | Data value as a character literal or a null value. |

**Note:**

1. In the catalog view, the value of COLVALUE is always shown in the database code page and can contain substitution characters. However, the statistics are gathered internally in the code page of the column’s table, and will therefore use actual column values when applied during query optimization.
Each row represents the distribution statistics that apply to the \( n \)th most frequent value of a column group or the \( n \)th quantile of a column group.

Table 92. SYSCAT.COLGROUPDISTCOUNTS Catalog View

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>COLGROUPID</td>
<td>INTEGER</td>
<td></td>
<td>Identifier for the column group.</td>
</tr>
<tr>
<td>TYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>• F = Frequency value</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Q = Quantile value</td>
</tr>
<tr>
<td>SEQNO</td>
<td>SMALLINT</td>
<td></td>
<td>Sequence number ( n ) representing the ( n )th TYPE value.</td>
</tr>
<tr>
<td>VALCOUNT</td>
<td>BIGINT</td>
<td></td>
<td>If ( ) = ‘F’, VALCOUNT is the number of occurrences of COLVALUE for the column group with this SEQNO. If ( ) = ‘Q’, VALCOUNT is the number of rows whose value is less than or equal to COLVALUE for the column group with this SEQNO.</td>
</tr>
<tr>
<td>DISTCOUNT</td>
<td>BIGINT</td>
<td></td>
<td>If ( ) = ‘Q’, this column records the number of distinct values that are less than or equal to COLVALUE for the column group with this SEQNO (the null value if unavailable).</td>
</tr>
</tbody>
</table>
SYSCAT.COLGROUPS

Each row represents a column group and statistics that apply to the entire column group.

Table 93. SYSCAT.COLGROUPS Catalog View

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>COLGROUPSCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the column group.</td>
</tr>
<tr>
<td>COLGROUPNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the column group.</td>
</tr>
<tr>
<td>COLGROUPID</td>
<td>INTEGER</td>
<td></td>
<td>Identifier for the column group.</td>
</tr>
<tr>
<td>COLGROUPCARD</td>
<td>BIGINT</td>
<td></td>
<td>Cardinality of the column group.</td>
</tr>
<tr>
<td>NUMFREQ_VALUES</td>
<td>SMALLINT</td>
<td></td>
<td>Number of frequent values collected for the column group.</td>
</tr>
<tr>
<td>NUMQUANTILES</td>
<td>SMALLINT</td>
<td></td>
<td>Number of quantiles collected for the column group.</td>
</tr>
</tbody>
</table>
Each row represents an identity column that is defined for a table.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TABSCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the table or view that contains the column.</td>
</tr>
<tr>
<td>TABNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the table or view that contains the column.</td>
</tr>
<tr>
<td>COLNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Name of the column.</td>
</tr>
<tr>
<td>START</td>
<td>DECIMAL (31,0)</td>
<td>Y</td>
<td>Start value of the sequence. The null value if the sequence is an alias.</td>
</tr>
<tr>
<td>INCREMENT</td>
<td>DECIMAL (31,0)</td>
<td>Y</td>
<td>Increment value. The null value if the sequence is an alias.</td>
</tr>
<tr>
<td>MINVALUE</td>
<td>DECIMAL (31,0)</td>
<td>Y</td>
<td>Minimum value of the sequence. The null value if the sequence is an alias.</td>
</tr>
<tr>
<td>MAXVALUE</td>
<td>DECIMAL (31,0)</td>
<td>Y</td>
<td>Maximum value of the sequence. The null value if the sequence is an alias.</td>
</tr>
<tr>
<td>CYCLE</td>
<td>CHAR (1)</td>
<td></td>
<td>Indicates whether or not the sequence can continue to generate values after reaching its maximum or minimum value.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = Sequence cannot cycle</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = Sequence can cycle</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Blank = Sequence is an alias.</td>
</tr>
<tr>
<td>CACHE</td>
<td>INTEGER</td>
<td></td>
<td>Number of sequence values to pre-allocate in memory for faster access. 0 indicates that values of the sequence are not to be preallocated. In a partitioned database, this value applies to each database partition. -1 if the sequence is an alias.</td>
</tr>
<tr>
<td>ORDER</td>
<td>CHAR (1)</td>
<td></td>
<td>Indicates whether or not the sequence numbers must be generated in order of request.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = Sequence numbers are not required to be generated in order of request</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = Sequence numbers must be generated in order of request</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Blank = Sequence is an alias.</td>
</tr>
<tr>
<td>NEXTCACHEFIRSTVALUE</td>
<td>DECIMAL (31,0)</td>
<td>Y</td>
<td>The first value available to be assigned in the next cache block. If no caching, the next value available to be assigned.</td>
</tr>
<tr>
<td>SEQID</td>
<td>INTEGER</td>
<td></td>
<td>Identifier for the sequence or alias.</td>
</tr>
</tbody>
</table>
Each row contains column-specific option values.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TABSCHEMA</td>
<td>VARCHAR (128)</td>
<td>Null</td>
<td>Schema name of the nickname.</td>
</tr>
<tr>
<td>TABNAME</td>
<td>VARCHAR (128)</td>
<td>Null</td>
<td>Nickname for the column for which options are set.</td>
</tr>
<tr>
<td>COLNAME</td>
<td>VARCHAR (128)</td>
<td>Null</td>
<td>Local column name.</td>
</tr>
<tr>
<td>OPTION</td>
<td>VARCHAR (128)</td>
<td>Null</td>
<td>Name of the column option.</td>
</tr>
<tr>
<td>SETTING</td>
<td>CLOB (32K)</td>
<td>Null</td>
<td>Value.</td>
</tr>
</tbody>
</table>
Each row represents a column defined for a table, view, or nickname.

**Table 96. SYSCAT.COLUMNS Catalog View**

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TABSCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the table, view, or nickname that contains the column.</td>
</tr>
<tr>
<td>TABNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the table, view, or nickname that contains the column.</td>
</tr>
<tr>
<td>COLNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Name of the column.</td>
</tr>
<tr>
<td>COLNO</td>
<td>SMALLINT</td>
<td></td>
<td>Number of this column in the table (starting with 0).</td>
</tr>
<tr>
<td>TYPESCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the data type for the column.</td>
</tr>
<tr>
<td>TYPENAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the data type for the column.</td>
</tr>
<tr>
<td>LENGTH</td>
<td>INTEGER</td>
<td></td>
<td>Maximum length of the data; 0 for distinct types. The LENGTH column indicates precision for DECIMAL fields, and indicates the number of bytes of storage required for decimal floating-point columns; that is, 8 and 16 for DECIMAL(16) and DECIMAL(34), respectively.</td>
</tr>
<tr>
<td>SCALE</td>
<td>SMALLINT</td>
<td></td>
<td>Scale if the column type is DECIMAL or number of digits of fractional seconds if the column type is TIMESTAMP; 0 otherwise.</td>
</tr>
<tr>
<td>DEFAULT</td>
<td>VARCHAR (254)</td>
<td>Y</td>
<td>Default value for the column of a table expressed as a constant, special register, or cast-function appropriate for the data type of the column. Can also be the keyword NULL. Values might be converted from what was specified as a default value. For example, date and time constants are shown in ISO format, cast-function names are qualified with schema names, and identifiers are delimited. Null value if a DEFAULT clause was not specified or the column is a view column.</td>
</tr>
<tr>
<td>NULLS</td>
<td>CHAR (1)</td>
<td></td>
<td>Nullability attribute for the column.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = Column is not nullable</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = Column is nullable</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>The value can be ‘N’ for a view column that is derived from an expression or function. Nevertheless, such a column allows null values when the statement using the view is processed with warnings for arithmetic errors.</td>
</tr>
<tr>
<td>CODEPAGE</td>
<td>SMALLINT</td>
<td></td>
<td>Code page used for data in this column; 0 if the column is defined as FOR BIT DATA or is not a string type.</td>
</tr>
</tbody>
</table>
### Table 96. SYSCAT.COLUMNS Catalog View (continued)

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>COLLATIONSCHEMA</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>For string types, the schema name of the collation for the column; the null value otherwise.</td>
</tr>
<tr>
<td>COLLATIONNAME</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>For string types, the unqualified name of the collation for the column; the null value otherwise.</td>
</tr>
<tr>
<td>LOGGED</td>
<td>CHAR (1)</td>
<td></td>
<td>Applies only to columns whose type is LOB or distinct based on LOB; blank otherwise.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = Column is not logged</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = Column is logged</td>
</tr>
<tr>
<td>COMPACT</td>
<td>CHAR (1)</td>
<td></td>
<td>Applies only to columns whose type is LOB or distinct based on LOB; blank otherwise.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = Column is not compacted</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = Column is compacted in storage</td>
</tr>
<tr>
<td>COLCARD</td>
<td>BIGINT</td>
<td></td>
<td>Number of distinct values in the column; -1 if statistics are not collected; -2 for inherited columns and columns of hierarchy tables.</td>
</tr>
<tr>
<td>HIGH2KEY³</td>
<td>VARCHAR (254)</td>
<td>Y</td>
<td>Second-highest data value. Representation of numeric data changed to character literals. Empty if statistics are not collected. Empty for inherited columns and columns of hierarchy tables.</td>
</tr>
<tr>
<td>LOW2KEY³</td>
<td>VARCHAR (254)</td>
<td>Y</td>
<td>Second-lowest data value. Representation of numeric data changed to character literals. Empty if statistics are not collected. Empty for inherited columns and columns of hierarchy tables.</td>
</tr>
<tr>
<td>AVGCOLLEN</td>
<td>INTEGER</td>
<td></td>
<td>Average space in bytes when the column is stored in database memory or a temporary table.  For LOB data types that are not inlined, LONG data types, and XML documents, the value used to calculate the average column length is the length of the data descriptor. An extra byte is required if the column is nullable; -1 if statistics have not been collected; -2 for inherited columns and columns of hierarchy tables. Note: The average space required to store the column on disk may be different than the value represented by this statistic.</td>
</tr>
<tr>
<td>KEYSEQ</td>
<td>SMALLINT</td>
<td>Y</td>
<td>The column’s numerical position within the table’s primary key. The null value for columns of subtables and hierarchy tables.</td>
</tr>
<tr>
<td>PARTKEYSEQ</td>
<td>SMALLINT</td>
<td>Y</td>
<td>The column’s numerical position within the table’s distribution key; 0 or the null value if the column is not in the distribution key. The null value for columns of subtables and hierarchy tables.</td>
</tr>
<tr>
<td>NQUANTILES</td>
<td>SMALLINT</td>
<td></td>
<td>Number of quantile values recorded in SYSCAT.COLDIST for this column; -1 if statistics are not gathered; -2 for inherited columns and columns of hierarchy tables.</td>
</tr>
</tbody>
</table>
### Table 96. SYSCAT.COLUMNS Catalog View (continued)

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NMOSTFREQ</td>
<td>SMALLINT</td>
<td></td>
<td>Number of most-frequent values recorded in SYSCAT.COLDIST for this column; -1 if statistics are not gathered; -2 for inherited columns and columns of hierarchy tables.</td>
</tr>
<tr>
<td>NUMNULLS</td>
<td>BIGINT</td>
<td></td>
<td>Number of null values in the column; -1 if statistics are not collected.</td>
</tr>
<tr>
<td>TARGETTYPESCHEMA</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>Schema name of the target row type, if the type of this column is REFERENCE; null value otherwise.</td>
</tr>
<tr>
<td>TARGETTYPENAME</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>Unqualified name of the target row type, if the type of this column is REFERENCE; null value otherwise.</td>
</tr>
<tr>
<td>SCOPE_TABSCHEMA</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>Schema name of the scope (target table), if the type of this column is REFERENCE; null value otherwise.</td>
</tr>
<tr>
<td>SCOPE_TABNAME</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>Unqualified name of the scope (target table), if the type of this column is REFERENCE; null value otherwise.</td>
</tr>
<tr>
<td>SOURCE_TABSCHEMA</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>For columns of typed tables or views, the schema name of the table or view in which the column was first introduced. For non-inherited columns, this is the same as TABSCHEMA. The null value for columns of non-typed tables and views.</td>
</tr>
<tr>
<td>SOURCE_TABNAME</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>For columns of typed tables or views, the unqualified name of the table or view in which the column was first introduced. For non-inherited columns, this is the same as TABNAME. The null value for columns of non-typed tables and views.</td>
</tr>
<tr>
<td>DLFEATURES</td>
<td>CHAR (10)</td>
<td>Y</td>
<td>This column is no longer used and will be removed in a future release.</td>
</tr>
<tr>
<td>SPECIAL_PROPS</td>
<td>CHAR (8)</td>
<td>Y</td>
<td>Applies to REFERENCE type columns only; blanks otherwise. Each byte position is defined as follows:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• 1 = Object identifier (OID) column ('Y' for yes; 'N' for no)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• 2 = User-generated or system-generated ('U' for user; 'S' for system)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bytes 3 through 8 are reserved for future use.</td>
</tr>
<tr>
<td>HIDDEN</td>
<td>CHAR (1)</td>
<td></td>
<td>Type of hidden column.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• 1 = Column is defined as IMPLICITLY HIDDEN</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• S = System-managed hidden column</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Blank = Column is not hidden</td>
</tr>
<tr>
<td>INLINE_LENGTH</td>
<td>INTEGER</td>
<td></td>
<td>Maximum size in bytes of the internal representation of an instance of an XML document, a structured type, or a LOB data type, that can be stored in the base table; 0 when not applicable.</td>
</tr>
<tr>
<td>Column Name</td>
<td>Data Type</td>
<td>Nullable</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>---------------</td>
<td>----------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>PCTINLINED</td>
<td>SMALLINT</td>
<td></td>
<td>Percentage of inlined XML documents or LOB data. -1 if statistics have not been collected.</td>
</tr>
</tbody>
</table>
| IDENTITY           | CHAR (1)      |          | • N = Not an identity column  
• T = Row change timestamp column  
• Y = Identity column |
| ROWCHANGETIMESTAMP | CHAR (1)      |          | • N = Not a row change timestamp column  
• Y = Row change timestamp column |
| GENERATED          | CHAR (1)      |          | Type of generated column.  
• A = Column value is always generated  
• D = Column value is generated by default  
• Blank = Column is not generated |
| TEXT               | CLOB (2M)     | Y        | For columns defined as generated as expression, this field contains the text of the generated column expression, starting with the keyword AS. |
| COMPRESS           | CHAR (1)      |          | • O = Compress off  
• S = Compress system default values |
| AVGDISTINCTPERPAGE | DOUBLE        | Y        | For future use. |
| PAGEVARIANCERATIO  | DOUBLE        | Y        | For future use. |
| SUB_COUNT          | SMALLINT      |          | Average number of sub-elements in the column. Applicable to character string columns only. |
| SUB_DELIM_LENGTH   | SMALLINT      |          | Average length of the delimiters that separate each sub-element in the column. Applicable to character string columns only. |
| AVGCOLLENCHAR      | INTEGER       |          | Average number of characters (based on the collation in effect for the column) required for the column; -1 if the data type of the column is long, LOB, or XML or if statistics have not been collected; -2 for inherited columns and columns of hierarchy tables. |
| IMPLICITVALUE      | VARCHAR (254) | Y        | For a column that was added to a table after the table was created, stores the default value at the time the column was added. For a column that was defined when the table was created, stores the null value. |
| SECLABELNAME        | VARCHAR(128)  | Y        | Name of the security label that is associated with the column if it is a protected column; the null value otherwise. |
| REMARKS             | VARCHAR (254) | Y        | User-provided comments, or the null value. |
Table 96. SYSCAT.COLUMNS Catalog View (continued)

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
</table>

Note:

1. For Version 2.1.0, cast-function names were not delimited and may still appear this way in the DEFAULT column. Also, some view columns included default values which will still appear in the DEFAULT column.

2. Starting with Version 2, value D (indicating not null with a default) is no longer used. Instead, use of WITH DEFAULT is indicated by a non-null value in the DEFAULT column.

3. In the catalog view, the values of HIGH2KEY and LOW2KEY are always shown in the database code page and can contain substitution characters. However, the statistics are gathered internally in the code page of the column’s table, and will therefore use actual column values when applied during query optimization.

4. Attaching a data partition is allowed unless IMPLICITVALUE for a specific column is a non-null value for both the source column and the target column, and the values do not match. In this case, you must drop the source table and then recreate it. A column can have a non-null value in the IMPLICITVALUE field if one of the following conditions is met:
   - The column is created as the result of an ALTER TABLE...ADD COLUMN statement
   - The IMPLICITVALUE field is propagated from a source table during attach
   - The IMPLICITVALUE field is inherited from a source table during detach
   - The IMPLICITVALUE field is set during database upgrade from Version 8 to Version 9, where it is determined to be an added column, or might be an added column. If the database is not certain whether the column is added or not, it is treated as added. An added column is a column that was created as the result of an ALTER TABLE...ADD COLUMN statement.

To avoid these inconsistencies during non-migration scenarios, it is recommended that you always create the tables that you are going to attach with all the columns already defined. That is, never use the ALTER TABLE statement to add columns to a table before attaching it.
Each row represents a column that is referenced in the DIMENSIONS clause of a CREATE TABLE statement.

**Table 97. SYSCAT.COLUSE Catalog View**

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TABSCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the table containing the column.</td>
</tr>
<tr>
<td>TABNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the table containing the column.</td>
</tr>
<tr>
<td>COLNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Name of the column.</td>
</tr>
<tr>
<td>DIMENSION</td>
<td>SMALLINT</td>
<td></td>
<td>Dimension number, based on the order of dimensions specified in the DIMENSIONS clause (initial position is 0). For a composite dimension, this value will be the same for each component of the dimension.</td>
</tr>
<tr>
<td>COLSEQ</td>
<td>SMALLINT</td>
<td></td>
<td>Numeric position of the column in the dimension to which it belongs (initial position is 0). The value is 0 for the single column in a noncomposite dimension.</td>
</tr>
<tr>
<td>TYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>Type of dimension.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• C = Clustering or multidimensional clustering</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• P = Partitioning</td>
</tr>
</tbody>
</table>
Each row represents a condition defined in a module.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONDSHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the condition.</td>
</tr>
<tr>
<td>CONDMODULENAME</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>Unqualified name of the module to which the condition belongs.</td>
</tr>
<tr>
<td>CONDNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the condition.</td>
</tr>
<tr>
<td>CONDID</td>
<td>INTEGER</td>
<td></td>
<td>Identifier for the condition.</td>
</tr>
<tr>
<td>CONDMODULEID</td>
<td>INTEGER</td>
<td>Y</td>
<td>Identifier of the module to which the condition belongs.</td>
</tr>
<tr>
<td>SQLSTATE</td>
<td>CHAR(5)</td>
<td>Y</td>
<td>SQLSTATE value associated with the condition.</td>
</tr>
<tr>
<td>OWNER</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Authorization ID of the owner of the condition.</td>
</tr>
<tr>
<td>OWNERTYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>• S = The owner is the system</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• U = The owner is an individual user</td>
</tr>
<tr>
<td>CREATE_TIME</td>
<td>TIMESTAMP</td>
<td></td>
<td>Time at which the condition was created.</td>
</tr>
<tr>
<td>REMARKS</td>
<td>VARCHAR (254)</td>
<td></td>
<td>User-provided comments, or the null value.</td>
</tr>
</tbody>
</table>
Each row represents a dependency of a constraint on some other object. The constraint depends on the object of type BTYPE of name BNAME, so a change to the object affects the constraint.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONSTNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the constraint.</td>
</tr>
<tr>
<td>TABSCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the table to which the constraint applies.</td>
</tr>
<tr>
<td>TABNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the table to which the constraint applies.</td>
</tr>
<tr>
<td>BTYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>Type of object on which the constraint depends. Possible values are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• F = Routine</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• I = Index</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• R = User-defined structured type</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• u = Module alias</td>
</tr>
<tr>
<td>BSHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the object on which the constraint depends.</td>
</tr>
<tr>
<td>BMODULENAME</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>Unqualified name of the module to which the object on which a dependency belongs. The null value if not a module object.</td>
</tr>
<tr>
<td>BNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the object on which the constraint depends.</td>
</tr>
<tr>
<td>BMODULEID</td>
<td>INTEGER</td>
<td>Y</td>
<td>Identifier for the module of the object on which the constraint depends.</td>
</tr>
</tbody>
</table>
SYSCAT.CONTEXTATTRIBUTES

Each row represents a trusted context attribute.

Table 100. SYSCAT.CONTEXTATTRIBUTES Catalog View

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONTEXTNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Name of the trusted context.</td>
</tr>
<tr>
<td>ATTR_NAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Name of the attribute. One of:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• ADDRESS</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• ENCRYPTION</td>
</tr>
<tr>
<td>ATTR_VALUE</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Value of the attribute.</td>
</tr>
<tr>
<td>ATTR_OPTIONS</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>If ATTR_NAME is ‘ADDRESS’, specifies the level of encryption required for this specific address. A null value indicates that the global ENCRYPTION attribute applies.</td>
</tr>
</tbody>
</table>
Each row represents a trusted context.

Table 101. SYSCAT.CONTEXTS Catalog View

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONTEXTNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Name of the trusted context.</td>
</tr>
<tr>
<td>CONTEXTID</td>
<td>INTEGER</td>
<td></td>
<td>Identifier for the trusted context.</td>
</tr>
<tr>
<td>SYSTEMAUTHID</td>
<td>VARCHAR (128)</td>
<td></td>
<td>The system authorization ID associated with the trusted context.</td>
</tr>
<tr>
<td>DEFAULTCONTEXTROLE</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>The default role for the context.</td>
</tr>
<tr>
<td>CREATE_TIME</td>
<td>TIMESTAMP</td>
<td></td>
<td>Time at which the trusted context was created.</td>
</tr>
<tr>
<td>ALTER_TIME</td>
<td>TIMESTAMP</td>
<td></td>
<td>Time at which the trusted context was last altered.</td>
</tr>
<tr>
<td>ENABLED</td>
<td>CHAR (1)</td>
<td></td>
<td>Trusted context state.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = Disabled</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = Enabled</td>
</tr>
<tr>
<td>AUDITPOLICYID</td>
<td>INTEGER</td>
<td>Y</td>
<td>Identifier for the audit policy.</td>
</tr>
<tr>
<td>AUDITPOLICYNAME</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>Name of the audit policy.</td>
</tr>
<tr>
<td>REMARKS</td>
<td>VARCHAR (254)</td>
<td>Y</td>
<td>User-provided comments, or the null value.</td>
</tr>
</tbody>
</table>
SYSCAT.DATAPARTITIONEXPRESSION

Each row represents an expression for that part of the table partitioning key.

Table 102. SYSCAT.DATAPARTITIONEXPRESSION Catalog View

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TABSCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the partitioned table.</td>
</tr>
<tr>
<td>TABNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the partitioned table.</td>
</tr>
<tr>
<td>DATAPARTITIONKEYSEQ</td>
<td>INTEGER</td>
<td></td>
<td>Expression key part sequence ID, starting from 1.</td>
</tr>
<tr>
<td>DATAPARTITIONEXPRESSION</td>
<td>CLOB (32K)</td>
<td></td>
<td>Expression for this entry in the sequence, in SQL syntax.</td>
</tr>
<tr>
<td>NULLSFIRST</td>
<td>CHAR (1)</td>
<td></td>
<td>• N = Null values in this expression compare high</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = Null values in this expression compare low</td>
</tr>
</tbody>
</table>
Each row represents a data partition. Note:

- The data partition statistics represent one database partition if the table is created on multiple database partitions.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATAPARTITIONNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Name of the data partition.</td>
</tr>
<tr>
<td>TABSCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the table to which this data partition belongs.</td>
</tr>
<tr>
<td>TABNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the table to which this data partition belongs.</td>
</tr>
<tr>
<td>DATAPARTITIONID</td>
<td>INTEGER</td>
<td>Y</td>
<td>Identifier for the data partition.</td>
</tr>
<tr>
<td>TBSPACEID</td>
<td>INTEGER</td>
<td>Y</td>
<td>Identifier for the table space in which this data partition is stored. The null value when STATUS is 'I'.</td>
</tr>
<tr>
<td>PARTITIONOBJECTID</td>
<td>INTEGER</td>
<td>Y</td>
<td>Identifier for the data partition within the table space.</td>
</tr>
<tr>
<td>LONG_TBSPACEID</td>
<td>INTEGER</td>
<td>Y</td>
<td>Identifier for the table space in which long data is stored. The null value when STATUS is 'I'.</td>
</tr>
<tr>
<td>ACCESS_MODE</td>
<td>CHAR (1)</td>
<td></td>
<td>Access restriction state of the data partition. These states only apply to objects that are in set integrity pending state or to objects that were processed by a SET INTEGRITY statement. Possible values are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• D = No data movement</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• F = Full access</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = No access</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• R = Read-only access</td>
</tr>
<tr>
<td>STATUS</td>
<td>VARCHAR (32)</td>
<td></td>
<td>• A = Data partition is newly attached</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• D = Data partition is detached and detached dependents are to be incrementally maintained with respect to the content of this partition</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• I = Detached data partition whose entry in the catalog is maintained only during asynchronous index cleanup; rows with a STATUS value of 'I' are removed when all index records referring to the detached partition have been deleted</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• L = Data partition is logically detached</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Empty string = Data partition is visible (normal status)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bytes 2 through 32 are reserved for future use.</td>
</tr>
<tr>
<td>SEQNO</td>
<td>INTEGER</td>
<td></td>
<td>Data partition sequence number (starting from 0).</td>
</tr>
<tr>
<td>LOWINCLUSIVE</td>
<td>CHAR (1)</td>
<td></td>
<td>• N = Low key value is not inclusive</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = Low key value is inclusive</td>
</tr>
<tr>
<td>Column Name</td>
<td>Data Type</td>
<td>Nullable</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>-------------------</td>
<td>----------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>LOWVALUE</td>
<td>VARCHAR (512)</td>
<td></td>
<td>Low key value (a string representation of an SQL value) for this data partition.</td>
</tr>
<tr>
<td>HIGHINCLUSIVE</td>
<td>CHAR (1)</td>
<td></td>
<td>• N = High key value is not inclusive</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = High key value is inclusive</td>
</tr>
<tr>
<td>HIGHVALUE</td>
<td>VARCHAR (512)</td>
<td></td>
<td>High key value (a string representation of an SQL value) for this data partition.</td>
</tr>
<tr>
<td>CARD</td>
<td>BIGINT</td>
<td></td>
<td>Total number of rows in the data partition; -1 if statistics are not collected.</td>
</tr>
<tr>
<td>OVERFLOW</td>
<td>BIGINT</td>
<td></td>
<td>Total number of overflow records in the data partition; -1 if statistics are not collected.</td>
</tr>
<tr>
<td>NPAGES</td>
<td>BIGINT</td>
<td></td>
<td>Total number of pages on which the rows of the data partition exist; -1 if statistics are not collected.</td>
</tr>
<tr>
<td>FPAGES</td>
<td>BIGINT</td>
<td></td>
<td>Total number of pages in the data partition; -1 if statistics are not collected.</td>
</tr>
<tr>
<td>ACTIVE_BLOCKS</td>
<td>BIGINT</td>
<td></td>
<td>Total number of active blocks in the data partition, or -1. Applies to multidimensional clustering (MDC) tables only.</td>
</tr>
<tr>
<td>INDEX_TBSPACEID</td>
<td>INTEGER</td>
<td></td>
<td>Identifier for the table space which holds all partitioned indexes for this data partition.</td>
</tr>
<tr>
<td>AVGRROWSIZE</td>
<td>SMALLINT</td>
<td></td>
<td>Average length (in bytes) of both compressed and uncompressed rows in this data partition; -1 if statistics are not collected.</td>
</tr>
<tr>
<td>PCTROWSCOMPRESSED</td>
<td>REAL</td>
<td></td>
<td>Compressed rows as a percentage of the total number of rows in the data partition; -1 if statistics are not collected.</td>
</tr>
<tr>
<td>PCTPAGESAVED</td>
<td>SMALLINT</td>
<td></td>
<td>Approximate percentage of pages saved in the data partition as a result of row compression. This value includes overhead bytes for each user data row in the data partition, but does not include the space that is consumed by dictionary overhead; -1 if statistics are not collected.</td>
</tr>
<tr>
<td>AVGCOMPRESSEDROWSIZE</td>
<td>SMALLINT</td>
<td></td>
<td>Average length (in bytes) of compressed rows in this data partition; -1 if statistics are not collected.</td>
</tr>
<tr>
<td>AVGROWCOMPRESSIONRATIO</td>
<td>REAL</td>
<td></td>
<td>For compressed rows in the data partition, this is the average compression ratio by row; that is, the average uncompressed row length divided by the average compressed row length; -1 if statistics are not collected.</td>
</tr>
<tr>
<td>STATS_TIME</td>
<td>TIMESTAMP</td>
<td>Y</td>
<td>Time at which any change was last made to recorded statistics for this object. Null if statistics are not collected.</td>
</tr>
<tr>
<td>LASTUSED</td>
<td>DATE</td>
<td></td>
<td>Date when the data partition was last used by any DML statement or the LOAD command. The default value is '0001-01-01'. This value is updated asynchronously.</td>
</tr>
</tbody>
</table>
Each row represents a dependency of a user-defined data type on some other object.

Table 104. SYSCAT.DATATYPEDEP Catalog View

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TYPESCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the data type.</td>
</tr>
<tr>
<td>TYPOFILENAME</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>Module name of the data type.</td>
</tr>
<tr>
<td>TYPENAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the data type.</td>
</tr>
<tr>
<td>TYPEMODULEID</td>
<td>INTEGER</td>
<td>Y</td>
<td>Identifier for the module of the data type.</td>
</tr>
<tr>
<td>BTYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>Type of object on which there is a dependency. Possible values are:</td>
</tr>
<tr>
<td>BSCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the object on which there is a dependency.</td>
</tr>
<tr>
<td>BMODULENAME</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>Module name of the object on which there is a dependency.</td>
</tr>
<tr>
<td>BNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the object on which there is a dependency.</td>
</tr>
<tr>
<td>BMODULEID</td>
<td>INTEGER</td>
<td>Y</td>
<td>Identifier for the module of the object on which there is a dependency.</td>
</tr>
<tr>
<td>TABAUTH</td>
<td>SMALLINT</td>
<td>Y</td>
<td>If BTYPE = 'S', 'T', 'U', 'V', 'W', or 'v', encodes the privileges on the table or view that are required by the dependent data type; the null value otherwise.</td>
</tr>
</tbody>
</table>
SYSCAT.DATATYPES

Each row represents a built-in or user-defined data type.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TYPESCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the data type if TYPEMODULEID is null; otherwise schema name of the module to which the data type belongs.</td>
</tr>
<tr>
<td>TYPEMODULENAME</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>Unqualified name of the module to which the user-defined type belongs. The null value if not a module user-defined type.</td>
</tr>
<tr>
<td>TYPENAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the data type.</td>
</tr>
<tr>
<td>OWNER</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Authorization ID of the owner of the type.</td>
</tr>
<tr>
<td>OWNERTYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>• S = The owner is the system</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• U = The owner is an individual user</td>
</tr>
<tr>
<td>SOURCESCHEMA</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>For distinct types or array types, the schema name of the source data type.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>For user-defined structured types, the schema name of the built-in type of the reference representation type. Null for other data types.</td>
</tr>
<tr>
<td>SOURCEMODULENAME</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>Unqualified name of the module to which the source data type belongs.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>The null value if not a module source data type.</td>
</tr>
<tr>
<td>SOURCENAME</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>For distinct types or array types, the unqualified name of the source data type. For user-defined structured types, the unqualified built-in type name of the reference representation type. Null for other data types.</td>
</tr>
<tr>
<td>METATYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>• A = User-defined array type</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• C = User-defined cursor type</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• F = User-defined row type</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• L = User-defined associative array type</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• R = User-defined structured type</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• S = System predefined type</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• T = User-defined distinct type</td>
</tr>
<tr>
<td>TYPEID</td>
<td>SMALLINT</td>
<td></td>
<td>Identifier for the data type.</td>
</tr>
<tr>
<td>TYPEMODULEID</td>
<td>INTEGER</td>
<td>Y</td>
<td>Identifier for the module to which the user-defined type belongs. The null value if not a module user-defined type.</td>
</tr>
<tr>
<td>SOURCETYPEID</td>
<td>SMALLINT</td>
<td>Y</td>
<td>Identifier for the source type (the null value for built-in types). For user-defined structured types, this is the identifier of the reference representation type.</td>
</tr>
<tr>
<td>SOURCEMODULEID</td>
<td>INTEGER</td>
<td>Y</td>
<td>Identifier for the module to which the source data type belongs. The null value if not a module source data type.</td>
</tr>
</tbody>
</table>
### Table 105. SYSCAT.DATATYPES Catalog View (continued)

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUBLISHED</td>
<td>CHAR (1)</td>
<td></td>
<td>Indicates whether the module user-defined type can be referenced outside its module.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = The module user-defined type is not published</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = The module user-defined type is published</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Blank = Not applicable</td>
</tr>
<tr>
<td>LENGTH</td>
<td>INTEGER</td>
<td></td>
<td>Maximum length of the type. 0 for built-in parameterized types (for example, DECIMAL and VARCHAR). For user-defined structured types, this is the length of the reference representation type.</td>
</tr>
<tr>
<td>SCALE</td>
<td>SMALLINT</td>
<td></td>
<td>Scale for distinct types or reference representation types based on the built-in DECIMAL type; the number of digits of fractional seconds for distinct types based on the built-in TIMESTAMP type; 6 for the built-in TIMESTAMP type; 0 for all other types (including DECIMAL itself).</td>
</tr>
<tr>
<td>CODEPAGE</td>
<td>SMALLINT</td>
<td></td>
<td>Database code page for string types, distinct types based on string types, or reference representation types; 0 otherwise.</td>
</tr>
<tr>
<td>COLLATIONSHEMA</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>For string types, the schema name of the collation for the data type; the null value otherwise.</td>
</tr>
<tr>
<td>COLLATIONNAME</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>For string types, the unqualified name of the collation for the data type; the null value otherwise.</td>
</tr>
<tr>
<td>ARRAY_LENGTH</td>
<td>INTEGER</td>
<td>Y</td>
<td>Maximum cardinality of the array. The null value if METATYPE is not ‘A’.</td>
</tr>
<tr>
<td>ARRAYINDEXTYPESHEMA</td>
<td>VARCHAR(128)</td>
<td>Y</td>
<td>Schema of the data type of the array index. The null value if METATYPE is not ‘L’.</td>
</tr>
<tr>
<td>ARRAYINDEXYPNAME</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>Name of the data type of the array index. The null value if METATYPE is not ‘L’.</td>
</tr>
<tr>
<td>ARRAYINDEXYPEID</td>
<td>SMALLINT</td>
<td>Y</td>
<td>Identifier for the array index type. The null value if METATYPE is not ‘L’.</td>
</tr>
<tr>
<td>ARRAYINDEXYPELENGTH</td>
<td>INTEGER</td>
<td>Y</td>
<td>Maximum length of the array index data type. The null value if METATYPE is not ‘L’.</td>
</tr>
<tr>
<td>CREATE_TIME</td>
<td>TIMESTAMP</td>
<td></td>
<td>Creation time of the data type.</td>
</tr>
<tr>
<td>VALID</td>
<td>CHAR (1)</td>
<td></td>
<td>• N = The data type is invalid</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = The data type is valid</td>
</tr>
<tr>
<td>ATTRCOUNT</td>
<td>SMALLINT</td>
<td></td>
<td>Number of attributes in the data type.</td>
</tr>
<tr>
<td>INSTANTIABLE</td>
<td>CHAR (1)</td>
<td></td>
<td>• N = Type cannot be instantiated</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = Type can be instantiated</td>
</tr>
<tr>
<td>WITH_FUNC_ACCESS</td>
<td>CHAR (1)</td>
<td></td>
<td>• N = Methods for this type cannot be invoked using function notation.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = All the methods for this type can be invoked using function notation.</td>
</tr>
</tbody>
</table>
### Table 105. SYSCAT.DATATYPES Catalog View (continued)

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FINAL</td>
<td>CHAR (1)</td>
<td><strong>Y</strong></td>
<td>- <strong>N</strong> = The user-defined type can have subtypes.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- <strong>Y</strong> = The user-defined type cannot have subtypes.</td>
</tr>
<tr>
<td>INLINE_LENGTH</td>
<td>INTEGER</td>
<td><strong>Y</strong></td>
<td>Maximum length of a structured type that can be kept with a base table row; 0 otherwise.</td>
</tr>
<tr>
<td>NATURAL_INLINE_LENGTH</td>
<td>INTEGER</td>
<td><strong>Y</strong></td>
<td>System-generated natural inline length of a structured type instance. The null value if this type is not a structured type.</td>
</tr>
<tr>
<td>JARSCHEMA</td>
<td>VARCHAR (128)</td>
<td><strong>Y</strong></td>
<td>Schema name of the JAR_ID that identifies the Jar file containing the Java class that implements the SQL type. The null value if the EXTERNAL NAME clause is not specified.</td>
</tr>
<tr>
<td>JAR_ID</td>
<td>VARCHAR (128)</td>
<td><strong>Y</strong></td>
<td>Identifier for the Jar file that contains the Java class that implements the SQL type. The null value if the EXTERNAL NAME clause is not specified.</td>
</tr>
<tr>
<td>CLASS</td>
<td>VARCHAR (384)</td>
<td><strong>Y</strong></td>
<td>Java class that implements the SQL type. The null value if the EXTERNAL NAME clause is not specified.</td>
</tr>
<tr>
<td>SQLJ_REPRESENTATION</td>
<td>CHAR (1)</td>
<td><strong>Y</strong></td>
<td>SQLJ &quot;representation_spec&quot; of the Java class that implements the SQL type. The null value if the EXTERNAL NAME LANGUAGE JAVA REPRESENTATION SPEC clause is not specified.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- <strong>D</strong> = SQL data</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- <strong>S</strong> = Serializable</td>
</tr>
<tr>
<td>ALTER_TIME</td>
<td>TIMESTAMP</td>
<td></td>
<td>Time at which the data type was last altered.</td>
</tr>
<tr>
<td>DEFINER?</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Authorization ID of the owner of the type.</td>
</tr>
<tr>
<td>REMARKS</td>
<td>VARCHAR (254)</td>
<td><strong>Y</strong></td>
<td>User-provided comments, or the null value.</td>
</tr>
</tbody>
</table>

**Note:**
1. The DEFINER column is included for backwards compatibility. See OWNER.
Each row represents a user, group, or role that has been granted one or more database-level authorities.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRANTOR</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Grantor of the authority.</td>
</tr>
<tr>
<td>GRANTORTYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>• S = Grantor is the system</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• U = Grantor is an individual user</td>
</tr>
<tr>
<td>GRANTEE</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Holder of the authority.</td>
</tr>
<tr>
<td>GRANTEEETYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>• G = Grantee is a group</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• R = Grantee is a role</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• U = Grantee is an individual user</td>
</tr>
<tr>
<td>BINDADDAUTH</td>
<td>CHAR (1)</td>
<td></td>
<td>Authority to create packages.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = Not held</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = Held</td>
</tr>
<tr>
<td>CONNECTAUTH</td>
<td>CHAR (1)</td>
<td></td>
<td>Authority to connect to the database.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = Not held</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = Held</td>
</tr>
<tr>
<td>CREATETABAUTH</td>
<td>CHAR (1)</td>
<td></td>
<td>Authority to create tables.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = Not held</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = Held</td>
</tr>
<tr>
<td>DBADMAUTH</td>
<td>CHAR (1)</td>
<td></td>
<td>DBADM authority.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = Not held</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = Held</td>
</tr>
<tr>
<td>EXTERNALROUTINEAUTH</td>
<td>CHAR (1)</td>
<td></td>
<td>Authority to create external routines.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = Not held</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = Held</td>
</tr>
<tr>
<td>IMPLSCHEMAAUTH</td>
<td>CHAR (1)</td>
<td></td>
<td>Authority to implicitly create schemas by creating objects in non-existent schemas.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = Not held</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = Held</td>
</tr>
<tr>
<td>LOADAUTH</td>
<td>CHAR (1)</td>
<td></td>
<td>Authority to use the DB2 load utility.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = Not held</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = Held</td>
</tr>
<tr>
<td>NOFENCEAUTH</td>
<td>CHAR (1)</td>
<td></td>
<td>Authority to create non-fenced user-defined functions.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = Not held</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = Held</td>
</tr>
<tr>
<td>QUIESCECONNECTAUTH</td>
<td>CHAR (1)</td>
<td></td>
<td>Authority to access the database when it is quiesced.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = Not held</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = Held</td>
</tr>
<tr>
<td>LIBRARYADMAUTH</td>
<td>CHAR (1)</td>
<td></td>
<td>Reserved for future use.</td>
</tr>
</tbody>
</table>
### SYSCAT.DBAUTH

**Table 106. SYSCAT.DBAUTH Catalog View (continued)**

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SECURITYADMAUTH</td>
<td>CHAR (1)</td>
<td></td>
<td>Authority to administer database security.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = Not held</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = Held</td>
</tr>
<tr>
<td>SQLADMAUTH</td>
<td>CHAR (1)</td>
<td></td>
<td>Authority to monitor and tune SQL statements.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = Not held</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = Held</td>
</tr>
<tr>
<td>WLMADMAUTH</td>
<td>CHAR (1)</td>
<td></td>
<td>Authority to manage WLM objects.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = Not held</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = Held</td>
</tr>
<tr>
<td>EXPLAINAUTH</td>
<td>CHAR (1)</td>
<td></td>
<td>Authority to explain SQL statements without requiring actual privileges on</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>the objects in the statement.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = Not held</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = Held</td>
</tr>
<tr>
<td>DATAACCESSAUTH</td>
<td>CHAR (1)</td>
<td></td>
<td>Authority to access data.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = Not held</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = Held</td>
</tr>
<tr>
<td>ACCESSCTRLAUTH</td>
<td>CHAR (1)</td>
<td></td>
<td>Authority to grant and revoke database object privileges.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = Not held</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = Held</td>
</tr>
</tbody>
</table>
Each row represents a database partition that is contained in a database partition group.

Table 107. SYSCAT.DBPARTITIONGROUPDEF Catalog View

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBPGNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Name of the database partition group that contains the database partition.</td>
</tr>
<tr>
<td>DBPARTITIONNUM</td>
<td>SMALLINT</td>
<td></td>
<td>Partition number of a database partition that is contained in the database partition group. A valid partition number is between 0 and 999, inclusive.</td>
</tr>
<tr>
<td>IN_USE</td>
<td>CHAR (1)</td>
<td></td>
<td>Status of the database partition.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• A = The newly added database partition is not in the distribution map, but the containers for the table spaces in the database partition group have been created; the database partition is added to the distribution map when a redistribute database partition group operation has completed successfully.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• D = The database partition will be dropped when a redistribute database partition group operation has completed successfully.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• T = The newly added database partition is not in the distribution map, and it was added using the WITHOUT TABLESPACES clause; containers must be added to the table spaces in the database partition group.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = The database partition is in the distribution map.</td>
</tr>
</tbody>
</table>
Each row represents a database partition group.

Table 108. SYSCAT.DBPARTITIONGROUPS Catalog View

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBPGNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Name of the database partition group.</td>
</tr>
<tr>
<td>OWNER</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Authorization ID of the owner of the database partition group.</td>
</tr>
<tr>
<td>OWNERSTYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>• S = The owner is the system</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• U = The owner is an individual user</td>
</tr>
<tr>
<td>PMAP_ID</td>
<td>SMALLINT</td>
<td></td>
<td>Identifier for the distribution map in the SYSCAT.PARTITIONMAPS catalog view.</td>
</tr>
<tr>
<td>REDISTRIBUTE_PMAP_ID</td>
<td>SMALLINT</td>
<td></td>
<td>Identifier for the distribution map currently being used for redistribution; -1 if redistribution is currently not in progress.</td>
</tr>
<tr>
<td>CREATE_TIME</td>
<td>TIMESTAMP</td>
<td></td>
<td>Creation time of the database partition group.</td>
</tr>
<tr>
<td>DEFINER1</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Authorization ID of the owner of the database partition group.</td>
</tr>
<tr>
<td>REMARKS</td>
<td>VARCHAR (254)</td>
<td>Y</td>
<td>User-provided comments, or the null value.</td>
</tr>
</tbody>
</table>

Note:
1. The DEFINER column is included for backwards compatibility. See OWNER.
## SYSCAT.EVENTMONITORS

Each row represents an event monitor.

### Table 109. SYSCAT.EVENTMONITORS Catalog View

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVMONNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Name of the event monitor.</td>
</tr>
<tr>
<td>OWNER</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Authorization ID of the owner of the event monitor.</td>
</tr>
</tbody>
</table>
| OWNERTYPE   | CHAR (1)    |          | • S = The owner is the system  
• U = The owner is an individual user |
| TARGET_TYPE | CHAR (1)    |          | Type of target to which event data is written.  
• F = File  
• P = Pipe  
• T = Table  
• U = Unformatted event table |
| TARGET      | VARCHAR (762) |          | Name of the target to which file or pipe event monitor data is written. For files, it can be either an absolute path name or a relative path name (relative to the database path for the database; this can be seen using the LIST ACTIVE DATABASES command). For pipes, it can be an absolute path name. |
| MAXFILES    | INTEGER    | Y        | Maximum number of event files that this event monitor permits in an event path. The null value if there is no maximum, or if TARGET_TYPE is not 'F' (file). |
| MAXFILESIZE | INTEGER    | Y        | Maximum size (in 4K pages) that each event file can attain before the event monitor creates a new file. The null value if there is no maximum, or if TARGET_TYPE is not 'F' (file). |
| BUFFERSIZE  | INTEGER    | Y        | Size of the buffer (in 4K pages) that is used by event monitors with file targets; null value otherwise. |
| IO_MODE     | CHAR (1)   | Y        | Mode of file input/output (I/O).  
• B = Blocked  
• N = Not blocked  
• Null value = TARGET_TYPE is not 'F' (file) or 'T' (table) |
| WRITE_MODE  | CHAR (1)   | Y        | Indicates how this event monitor handles existing event data when the monitor is activated.  
• A = Append  
• R = Replace  
• Null value = TARGET_TYPE is not 'F' (file) |
### SYSCAT.EVENTMONITORS

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
</table>
| AUTOSTART    | CHAR (1)  |          | Indicates whether this event monitor is to be activated automatically when the database starts.  
• N = No  
• Y = Yes     |
| DBPARTITIONNUM | SMALLINT |          | Number of the database partition where the event monitor runs and logs events. |
| MONSCOPE     | CHAR (1)  |          | Monitoring scope.  
• G = Global  
• L = Local  
• T = Each database partition on which the table space exists  
• Blank = WRITE TO TABLE event monitor |
| EVMON_ACTIVATES | INTEGER |          | Number of times the event monitor has been activated.                       |
| NODENUM¹     | SMALLINT  |          | Number of the database partition where the event monitor runs and logs events. |
| DEFINER²     | VARCHAR (128) |          | Authorization ID of the owner of the event monitor.                         |
| REMARKS      | VARCHAR (254) | Y         | Reserved for future use.                                                   |

**Note:**

1. The NODENUM column is included for backwards compatibility. See DBPARTITIONNUM.
2. The DEFINER column is included for backwards compatibility. See OWNER.
SYSCAT.EVENTS

Each row represents an event that is being monitored. An event monitor, in general, monitors multiple events.

**Table 110. SYSCAT.EVENTS Catalog View**

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVMONNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Name of the event monitor that is monitoring this event.</td>
</tr>
<tr>
<td>TYPE</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Type of event being monitored. Possible values are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• ACTIVITIES</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• CONNECTIONS</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• DATABASE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• DEADLOCKS</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• DETAILDEADLOCKS</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• LOCKING</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• PKGCACHEBASE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• PKGCACHEDETAILED</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• STATEMENTS</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• TABLES</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• TABLESPACES</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• THRESHOLDVIOLATIONS</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• TRANSACTIONS</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• STATISTICS</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• UOW</td>
</tr>
<tr>
<td>FILTER</td>
<td>CLOB (64K)</td>
<td>Y</td>
<td>Full text of the WHERE clause that applies to this event.</td>
</tr>
</tbody>
</table>
Each row represents the target table of an event monitor that writes to SQL tables.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVMONNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Name of the event monitor.</td>
</tr>
<tr>
<td>LOGICAL_GROUP</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Name of the logical data group. Possible values are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• ACTIVITYHISTORY</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• BUFFERPOOL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• CONN</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• CONNHEADER</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• CONTROL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• DATAVAL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• DB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• DEADLOCK</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• DLCONN</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• DLLOCK</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• LOCKING</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• PKGCACHEBASE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• PKGCACHEDETAILED</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• SCSTATS</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• STMT</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• STMTHIST</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• STMTVALS</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• SUBSECTION</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• TABLE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• TABLESPACE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• THRESHOLDVIOLATIONS</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• UOW</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• WCSTATS</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• WLSTATS</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• XACT</td>
</tr>
<tr>
<td>TABSCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the target table.</td>
</tr>
<tr>
<td>TABNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the target table.</td>
</tr>
<tr>
<td>PCTDEACTIVATE</td>
<td>SMALLINT</td>
<td></td>
<td>A percent value that specifies how full a DMS table space must be before an</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>event monitor automatically deactivates. Set to 100 for SMS table spaces.</td>
</tr>
</tbody>
</table>
Each row represents the relationship between a subtable and a supertable, a subtype and a supertype, or a subview and a superview. All hierarchical relationships, including immediate ones, are included in this view.

Table 112. SYSCAT.FULLHIERARCHIES Catalog View

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>METATYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>Relationship type.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• R = Between structured types</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• U = Between typed tables</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• W = Between typed views</td>
</tr>
<tr>
<td>SUB_SCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the subtype, subtable, or subview.</td>
</tr>
<tr>
<td>SUB_NAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the subtype, subtable, or subview.</td>
</tr>
<tr>
<td>SUPER_SCHEMA</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>Schema name of the supertype, supertable, or superview.</td>
</tr>
<tr>
<td>SUPER_NAME</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>Unqualified name of the supertype, supertable, or superview.</td>
</tr>
<tr>
<td>ROOT_SCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the table, view, or type that is at the root of the hierarchy.</td>
</tr>
<tr>
<td>ROOT_NAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the table, view, or type that is at the root of the hierarchy.</td>
</tr>
</tbody>
</table>
SYSCAT.FUNCMAPOPTIONS

Each row represents a function mapping option value.

Table 113. SYSCAT.FUNCMAPOPTIONS Catalog View

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FUNCTION_MAPPING</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Name of the function mapping.</td>
</tr>
<tr>
<td>OPTION</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Name of the function mapping option.</td>
</tr>
<tr>
<td>SETTING</td>
<td>VARCHAR (2048)</td>
<td></td>
<td>Value of the function mapping option.</td>
</tr>
</tbody>
</table>
SYSCAT.FUNCMAPPARMOPTIONS

Each row represents a function mapping parameter option value.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FUNCTION_MAPPING</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Name of the function mapping.</td>
</tr>
<tr>
<td>ORDINAL</td>
<td>SMALLINT</td>
<td></td>
<td>Position of the parameter.</td>
</tr>
<tr>
<td>LOCATION</td>
<td>CHAR (1)</td>
<td></td>
<td>Location of the parameter.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• L = Local parameter</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• R = Remote parameter</td>
</tr>
<tr>
<td>OPTION</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Name of the function mapping parameter option.</td>
</tr>
<tr>
<td>SETTING</td>
<td>VARCHAR (2048)</td>
<td></td>
<td>Value of the function mapping parameter option.</td>
</tr>
</tbody>
</table>
Each row represents a function mapping.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FUNCTION_MAPPING</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Name of the function mapping (might be system-generated).</td>
</tr>
<tr>
<td>FUNCSCHEMA</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>Schema name of the function. If the null value, the function is assumed to be a built-in function.</td>
</tr>
<tr>
<td>FUNCNAME</td>
<td>VARCHAR (1024)</td>
<td>Y</td>
<td>Unqualified name of the user-defined or built-in function.</td>
</tr>
<tr>
<td>FUNCID</td>
<td>INTEGER</td>
<td>Y</td>
<td>Identifier for the function.</td>
</tr>
<tr>
<td>SPECIFICNAME</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>Name of the routine instance (might be system-generated).</td>
</tr>
<tr>
<td>OWNER</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Authorization ID of the owner of the mapping. ‘SYSIBM’ indicates that this is a built-in function.</td>
</tr>
</tbody>
</table>
| OWNTYPE             | CHAR (1)   |          | • S = The owner is the system  
• U = The owner is an individual user                                           |
| WRAPNAME            | VARCHAR (128) | Y        | Wrapper to which this mapping applies.                                     |
| SERVERNAME          | VARCHAR (128) | Y        | Name of the data source.                                                   |
| SERVERTYPE          | VARCHAR (30) | Y        | Type of data source to which this mapping applies.                         |
| SERVERVERSION       | VARCHAR (18) | Y        | Version of the server type to which this mapping applies.                  |
| CREATE_TIME         | TIMESTAMP  |          | Time at which the mapping was created.                                     |
| DEFINER             | VARCHAR (128) |         | Authorization ID of the owner of the mapping. ‘SYSIBM’ indicates that this is a built-in function. |
| REMARKS             | VARCHAR (254) | Y        | User-provided comments, or the null value.                                |

Note:
1. The DEFINER column is included for backwards compatibility. See OWNER.
SYSCAT.HIERARCHIES

Each row represents the relationship between a subtable and its immediate supertable, a subtype and its immediate supertype, or a subview and its immediate superview. Only immediate hierarchical relationships are included in this view.

Table 116. SYSCAT.HIERARCHIES Catalog View

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>METATYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>Relationship type.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• R = Between structured types</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• U = Between typed tables</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• W = Between typed views</td>
</tr>
<tr>
<td>SUB_SCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the subtype, subtable, or subview.</td>
</tr>
<tr>
<td>SUB_NAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the subtype, subtable, or subview.</td>
</tr>
<tr>
<td>SUPER_SCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the supertype, supertable, or superview.</td>
</tr>
<tr>
<td>SUPER_NAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the supertype, supertable, or superview.</td>
</tr>
<tr>
<td>ROOT_SCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the table, view, or type that is at the root of the hierarchy.</td>
</tr>
<tr>
<td>ROOT_NAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the table, view, or type that is at the root of the hierarchy.</td>
</tr>
</tbody>
</table>
Each row represents a histogram template bin.

Table 117. SYSCAT.HISTOGRAMTEMPLATEBINS Catalog View

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEMPLATENAME</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>Name of the histogram template.</td>
</tr>
<tr>
<td>TEMPLATEID</td>
<td>INTEGER</td>
<td></td>
<td>Identifier for the histogram template.</td>
</tr>
<tr>
<td>BINID</td>
<td>INTEGER</td>
<td></td>
<td>Identifier for the histogram template bin.</td>
</tr>
<tr>
<td>BINUPPERVALUE</td>
<td>BIGINT</td>
<td></td>
<td>The upper value for a single bin in the histogram template.</td>
</tr>
</tbody>
</table>
SYSCAT.HISTOGRAMTEMPLATES

Each row represents a histogram template.

**Table 118. SYSCAT.HISTOGRAMTEMPLATES Catalog View**

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEMPLATEID</td>
<td>INTEGER</td>
<td></td>
<td>Identifier for the histogram template.</td>
</tr>
<tr>
<td>TEMPLATENAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Name of the histogram template.</td>
</tr>
<tr>
<td>CREATE_TIME</td>
<td>TIMESTAMP</td>
<td></td>
<td>Time at which the histogram template was created.</td>
</tr>
<tr>
<td>ALTER_TIME</td>
<td>TIMESTAMP</td>
<td></td>
<td>Time at which the histogram template was last altered.</td>
</tr>
<tr>
<td>NUMBINS</td>
<td>INTEGER</td>
<td></td>
<td>Number of bins in the histogram template, including the last bin that has an unbounded top value.</td>
</tr>
<tr>
<td>REMARKS</td>
<td>VARCHAR (254)</td>
<td>Y</td>
<td>User-provided comments, or the null value.</td>
</tr>
</tbody>
</table>
Each row represents a relationship between a workload management object that can use histogram templates and a histogram template.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEMPLATENAME</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>Name of the histogram template.</td>
</tr>
<tr>
<td>TEMPLATEID</td>
<td>INTEGER</td>
<td></td>
<td>Identifier for the histogram template.</td>
</tr>
<tr>
<td>HISTOGRAMTYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>The type of information collected by histograms based on this template.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• C = Activity estimated cost histogram</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• E = Activity execution time histogram</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• I = Activity interarrival time histogram</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• L = Activity life time histogram</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Q = Activity queue time histogram</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• R = Request execution time histogram</td>
</tr>
<tr>
<td>OBJECTTYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>The type of WLM object.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• b = Service class</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• k = Work action</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• w = Workload</td>
</tr>
<tr>
<td>OBJECTID</td>
<td>INTEGER</td>
<td></td>
<td>Identifier of the WLM object.</td>
</tr>
<tr>
<td>SERVICECLASSNAME</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>Name of the service class.</td>
</tr>
<tr>
<td>PARENTSERVICECLASSNAME</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>The name of the parent service class of the service subclass that uses the histogram template.</td>
</tr>
<tr>
<td>WORKACTIONNAME</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>The name of the work action that uses the histogram template.</td>
</tr>
<tr>
<td>WORKACTIONSETNAME</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>The name of the work action set containing the work action that uses the histogram template.</td>
</tr>
<tr>
<td>WORKLOADNAME</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>The name of the workload that uses the histogram template.</td>
</tr>
</tbody>
</table>
SYSCAT.INDEXAUTH

Each row represents a user, group, or role that has been granted CONTROL privilege on an index.

Table 120. SYSCAT.INDEXAUTH Catalog View

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRANTOR</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Grantor of the privilege.</td>
</tr>
<tr>
<td>GRANTORTYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>• S = Grantor is the system</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• U = Grantor is an individual user</td>
</tr>
<tr>
<td>GRANTEE</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Holder of the privilege.</td>
</tr>
<tr>
<td>GRANTEEETYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>• G = Grantee is a group</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• R = Grantee is a role</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• U = Grantee is an individual user</td>
</tr>
<tr>
<td>INDSCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the index.</td>
</tr>
<tr>
<td>INDNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the index.</td>
</tr>
<tr>
<td>CONTROLAUTH</td>
<td>CHAR (1)</td>
<td></td>
<td>CONTROL privilege.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = Not held</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = Held</td>
</tr>
</tbody>
</table>
Each row represents a column that participates in an index.

Table 121. SYSCAT.INDEXCOLUSE Catalog View

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INDSHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the index.</td>
</tr>
<tr>
<td>INDNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the index.</td>
</tr>
<tr>
<td>COLNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Name of the column.</td>
</tr>
<tr>
<td>COLSEQ</td>
<td>SMALLINT</td>
<td></td>
<td>Numeric position of the column in the index (initial position is 1).</td>
</tr>
</tbody>
</table>
| COLORORDER    | CHAR (1)        |          | Order of the values in this index column. Possible values are: 
|               |                 |          | • A = Ascending                                                              |
|               |                 |          | • D = Descending                                                             |
|               |                 |          | • I = INCLUDE column (ordering ignored)                                       |
| COLLATIONSCHEMA| VARCHAR (128)   | Y        | For string types, the schema name of the collation for the column; the null value otherwise. |
| COLLATIONNAME | VARCHAR (128)   | Y        | For string types, the unqualified name of the collation for the column; the null value otherwise. |
SYSCAT.INDEXDEP

Each row represents a dependency of an index on some other object. The index depends on an object of type BTYPE and name BNAME, so a change to the object affects the index.

Table 122. SYSCAT.INDEXDEP Catalog View

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INDSCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the index.</td>
</tr>
<tr>
<td>INDNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the index.</td>
</tr>
<tr>
<td>BTYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>Type of object on which there is a dependency. Possible values are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• A = Table alias</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• B = Trigger</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• F = Routine</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• G = Global temporary table</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• H = Hierarchy table</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• K = Package</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• L = Detached table</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = Nickname</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• O = Privilege dependency on all subtables or subviews in a table or view hierarchy</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Q = Sequence</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• R = User-defined data type</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• S = Materialized query table</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• T = Table (not typed)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• U = Typed table</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• V = View (not typed)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• W = Typed view</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• X = Index extension</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Z = XSR object</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• q = Sequence alias</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• u = Module alias</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• v = Global variable</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• * = Anchored to the row of a base table</td>
</tr>
<tr>
<td>BSCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the object on which there is a dependency.</td>
</tr>
<tr>
<td>BMODULENAME</td>
<td>VARCHAR(128)</td>
<td>Y</td>
<td>Unqualified name of the module to which the object on which a dependency belongs. The null value if not a module object.</td>
</tr>
<tr>
<td>BNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the object on which there is a dependency. For routines (BTYPE = 'F'), this is the specific name.</td>
</tr>
<tr>
<td>BMODULEID</td>
<td>INTEGER</td>
<td>Y</td>
<td>Identifier for the module of the object on which there is a dependency.</td>
</tr>
<tr>
<td>TABAUTH</td>
<td>SMALLINT</td>
<td>Y</td>
<td>If BTYPE = 'O', 'S', 'T', 'U', 'V', 'W', or 'v', encodes the privileges on the table or view that are required by the dependent index; the null value otherwise.</td>
</tr>
</tbody>
</table>
SYSCAT.INDEXES

Each row represents an index. Indexes on typed tables are represented by two rows: one for the "logical index" on the typed table, and one for the "H-index" on the hierarchy table.

Table 123. SYSCAT.INDEXES Catalog View

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INDSHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the index.</td>
</tr>
<tr>
<td>INDNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the index.</td>
</tr>
<tr>
<td>OWNER</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Authorization ID of the owner of the index.</td>
</tr>
<tr>
<td>OWNERTYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>• S  = The owner is the system</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• U  = The owner is an individual user</td>
</tr>
<tr>
<td>TABSCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the table or nickname on which the index is defined.</td>
</tr>
<tr>
<td>TABNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the table or nickname on which the index is defined.</td>
</tr>
<tr>
<td>COLNAMES</td>
<td>VARCHAR (640)</td>
<td></td>
<td>This column is no longer used and will be removed in the next release. Use</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>SYSCAT.INDEXCOLUSE for this information.</td>
</tr>
<tr>
<td>UNIQUERULE</td>
<td>CHAR (1)</td>
<td></td>
<td>Unique rule.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• D  = Permits duplicates</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• U  = Unique</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• P  = Implements primary key</td>
</tr>
<tr>
<td>MADE_UNIQUE</td>
<td>CHAR (1)</td>
<td></td>
<td>• N  = Index remains as it was created</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y  = This index was originally non-unique but was converted to a unique</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>index to support a unique or primary key constraint. If the constraint is</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>dropped, the index reverts to being non-unique.</td>
</tr>
<tr>
<td>COLCOUNT</td>
<td>SMALLINT</td>
<td></td>
<td>Number of columns in the key, plus the number of include columns, if any.</td>
</tr>
<tr>
<td>UNIQUE_COLCOUNT</td>
<td>SMALLINT</td>
<td></td>
<td>Number of columns required for a unique key. It is always &lt;= COLCOUNT, and &lt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>COLCOUNT only if there are include columns; -1 if the index has no unique</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>key (that is, it permits duplicates).</td>
</tr>
<tr>
<td>INDEXTYPE</td>
<td>CHAR (4)</td>
<td></td>
<td>Type of index.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• BLOK = Block index</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• CLUS = Clustering index (controls the physical placement of newly inserted</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>rows)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• DIM = Dimension block index</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• REG = Regular index</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• XPTH = XML path index</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• XRGN = XML region index</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• XVIL = Index over XML column (logical)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• XVIP = Index over XML column (physical)</td>
</tr>
</tbody>
</table>
Table 123. SYSCAT.INDEXES Catalog View (continued)

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
</table>
| ENTRYTYPE   | CHAR (1)  | v        | H = This row represents an index on a hierarchy table  
|             |           |          | L = This row represents a logical index on a typed table  
|             |           |          | Blank = This row represents an index on an untyped table |
| PCTFREE     | SMALLINT  |          | Percentage of each index page to be reserved during the initial building of the index. This space is available for data insertions after the index has been built. |
| IID         | SMALLINT  |          | Identifier for the index. |
| NLEAF       | BIGINT    |          | Number of leaf pages; -1 if statistics are not collected. |
| NLEVELS     | SMALLINT  |          | Number of index levels; -1 if statistics are not collected. |
| FIRSTKEYCARD| BIGINT    |          | Number of distinct first-key values; -1 if statistics are not collected, or if not applicable. |
| FIRST2KEYCARD| BIGINT  |          | Number of distinct keys using the first two columns of the index; -1 if statistics are not collected, or if not applicable. |
| FIRST3KEYCARD| BIGINT |          | Number of distinct keys using the first three columns of the index; -1 if statistics are not collected, or if not applicable. |
| FIRST4KEYCARD| BIGINT |          | Number of distinct keys using the first four columns of the index; -1 if statistics are not collected, or if not applicable. |
| FULLKEYCARD | BIGINT    |          | Number of distinct full-key values; -1 if statistics are not collected. |
| CLUSTERRATIO³ | SMALLINT |          | Degree of data clustering with the index; -1 if statistics are not collected or if detailed index statistics are collected (in which case, CLUSTERFACTOR will be used instead). |
| CLUSTERFACTOR³ | DOUBLE |          | Finer measurement of the degree of clustering; -1 if statistics are not collected or if the index is defined on a nickname. |
| SEQUENTIAL_PAGES | BIGINT |          | Number of leaf pages located on disk in index key order with few or no large gaps between them; -1 if statistics are not collected. |
| DENSITY     | INTEGER   |          | Ratio of SEQUENTIAL_PAGES to number of pages in the range of pages occupied by the index, expressed as a percent (integer between 0 and 100); -1 if statistics are not collected. |
| USER_DEFINED | SMALLINT |          | 1 if this index was defined by a user and has not been dropped; 0 otherwise. |
### Table 123. SYSCAT.INDEXES Catalog View (continued)

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYSTEM_REQUIRED</td>
<td>SMALLINT</td>
<td></td>
<td>• 1 if one or the other of the following conditions is met:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- This index is required for a primary or unique key constraint, or this index is a dimension block index or composite block index for a multidimensional clustering (MDC) table.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- This is the index on the object identifier (OID) column of a typed table.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Y</td>
<td>• 2 if both of the following conditions are met:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- This index is required for a primary or unique key constraint, or this index is a dimension block index or composite block index for an MDC table.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- This is the index on the OID column of a typed table.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>• 0 otherwise.</td>
</tr>
<tr>
<td>CREATE_TIME</td>
<td>TIMESTAMP</td>
<td></td>
<td>Time when the index was created.</td>
</tr>
<tr>
<td>STATS_TIME</td>
<td>TIMESTAMP</td>
<td>Y</td>
<td>Last time that any change was made to the recorded statistics for this index. The null value if no statistics are available.</td>
</tr>
<tr>
<td>PAGE_FETCH_PAIRS</td>
<td>VARCHAR (520)</td>
<td></td>
<td>A list of pairs of integers, represented in character form. Each pair represents the number of pages in a hypothetical buffer, and the number of page fetches required to scan the table with this index using that hypothetical buffer. Zero-length string if no data is available.</td>
</tr>
<tr>
<td>MINPCTUSED</td>
<td>SMALLINT</td>
<td></td>
<td>A non-zero integer value indicates that the index is enabled for online defragmentation, and represents the minimum percentage of used space on a page before a page merge can be attempted. A zero value indicates that no page merge is attempted.</td>
</tr>
<tr>
<td>REVERSE_SCANS</td>
<td>CHAR (1)</td>
<td></td>
<td>• N = Index does not support reverse scans</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = Index supports reverse scans</td>
</tr>
<tr>
<td>INTERNAL_FORMAT</td>
<td>SMALLINT</td>
<td></td>
<td>Possible values are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• 1 = Index does not have backward pointers</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• 2 or greater = Index has backward pointers</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• 6 = Index is a composite block index</td>
</tr>
<tr>
<td>COMPRESSION</td>
<td>CHAR (1)</td>
<td></td>
<td>Specifies whether index compression is activated</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = Not activated</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = Activated</td>
</tr>
<tr>
<td>IESCHEMA</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>Schema name of the index extension. The null value for ordinary indexes.</td>
</tr>
<tr>
<td>Column Name</td>
<td>Data Type</td>
<td>Nullable</td>
<td>Description</td>
</tr>
<tr>
<td>------------------</td>
<td>-----------------</td>
<td>----------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>IENAME</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>Unqualified name of the index extension. The null value for ordinary indexes.</td>
</tr>
<tr>
<td>IEARGUMENTS</td>
<td>CLOB (64K)</td>
<td>Y</td>
<td>External information of the parameter specified when the index is created. The null value for ordinary indexes.</td>
</tr>
<tr>
<td>INDEX_OBJECTID</td>
<td>INTEGER</td>
<td></td>
<td>Identifier for the index object.</td>
</tr>
<tr>
<td>NUMRIDS</td>
<td>BIGINT</td>
<td></td>
<td>Total number of row identifiers (RIDs) or block identifiers (BIDs) in the index; -1 if not known.</td>
</tr>
<tr>
<td>NUMRIDS_DELETED</td>
<td>BIGINT</td>
<td></td>
<td>Total number of row identifiers (or block identifiers) in the index that are marked deleted, excluding those identifiers on leaf pages on which all the identifiers are marked deleted.</td>
</tr>
<tr>
<td>NUM_EMPTY_LEAPS</td>
<td>BIGINT</td>
<td></td>
<td>Total number of index leaf pages that have all of their row identifiers (or block identifiers) marked deleted.</td>
</tr>
<tr>
<td>AVERAGE_RANDOM_FETCH_PAGES</td>
<td>DOUBLE</td>
<td></td>
<td>Average number of random table pages between sequential page accesses when fetching using the index; -1 if not known.</td>
</tr>
<tr>
<td>AVERAGE_RANDOM_PAGES</td>
<td>DOUBLE</td>
<td></td>
<td>Average number of random table pages between sequential page accesses; -1 if not known.</td>
</tr>
<tr>
<td>AVERAGE_SEQUENCE_GAP</td>
<td>DOUBLE</td>
<td></td>
<td>Gap between index page sequences. Detected through a scan of index leaf pages, each gap represents the average number of index pages that must be randomly fetched between sequences of index pages; -1 if not known.</td>
</tr>
<tr>
<td>AVERAGE_SEQUENCE_FETCH_GAP</td>
<td>DOUBLE</td>
<td></td>
<td>Gap between table page sequences when fetching using the index. Detected through a scan of index leaf pages, each gap represents the average number of table pages that must be randomly fetched between sequences of table pages; -1 if not known.</td>
</tr>
<tr>
<td>TBSPACEID</td>
<td>INTEGER</td>
<td></td>
<td>Identifier for the index table space.</td>
</tr>
<tr>
<td>LEVEL2PCTFREE</td>
<td>SMALLINT</td>
<td></td>
<td>Percentage of each index level 2 page to be reserved during initial building of the index. This space is available for future inserts after the index has been built.</td>
</tr>
</tbody>
</table>
### Table 123. SYSCAT.INDEXES Catalog View (continued)

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAGESPLIT</td>
<td>CHAR (1)</td>
<td></td>
<td>Index page split behavior.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• H = High</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• L = Low</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• S = Symmetric</td>
</tr>
<tr>
<td>AVGPARTITION_</td>
<td>SMALLINT</td>
<td></td>
<td>Degree of data clustering within a single data partition. -1 if the table is not partitioned, if statistics are not collected, or if detailed statistics are collected (in which case AVGPARTITION_ CLUSTERFACTOR will be used instead).</td>
</tr>
<tr>
<td>CLUSTER RATIO³</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AVGPARTITION_</td>
<td>DOUBLE</td>
<td></td>
<td>Finer measurement of the degree of clustering within a single data partition. -1 if the table is not partitioned, if statistics are not collected, or if the index is defined on a nickname.</td>
</tr>
<tr>
<td>CLUSTERFACTOR³</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AVGPARTITION_PAGE_FETCH_</td>
<td>VARCHAR (520)</td>
<td></td>
<td>A list of paired integers in character form. Each pair represents a potential buffer pool size and the corresponding page fetches required to access a single data partition from the table. Zero-length string if no data is available, or if the table is not partitioned.</td>
</tr>
<tr>
<td>PAIRS³</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PCTPAGESAVED</td>
<td>SMALLINT</td>
<td></td>
<td>Approximate percentage of pages saved in the index as a result of index compression. -1 if statistics are not collected.</td>
</tr>
<tr>
<td>DATAPARTITION_</td>
<td>DOUBLE</td>
<td></td>
<td>A statistic measuring the &quot;clustering&quot; of the index keys with regard to data partitions. It is a number between 0 and 1, with 1 representing perfect clustering and 0 representing no clustering.</td>
</tr>
<tr>
<td>CLUSTERFACTOR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INDCARD</td>
<td>BIGINT</td>
<td></td>
<td>Cardinality of the index. This might be different from the cardinality of the table for indexes that do not have a one-to-one relationship between the table rows and the index entries.</td>
</tr>
<tr>
<td>AVERAGEKEYSIZE</td>
<td>INTEGER</td>
<td></td>
<td>Average index key size for keys on leaf pages in the index.</td>
</tr>
<tr>
<td>AVGNLEAFKEYSIZE</td>
<td>INTEGER</td>
<td></td>
<td>Average index key size for keys on non-leaf pages in the index.</td>
</tr>
<tr>
<td>OS_PTR_SIZE</td>
<td>INTEGER</td>
<td></td>
<td>Platform word size with which the index was created.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• 32 = 32-bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• 64 = 64-bit</td>
</tr>
<tr>
<td>COLLECTSTATISTICS</td>
<td>CHAR (1)</td>
<td></td>
<td>Specifies how statistics were collected at index creation time.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• D = Collect detailed index statistics</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• S = Collect sampled detailed index statistics</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = Collect basic index statistics</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Blank = Do not collect index statistics</td>
</tr>
<tr>
<td>DEFINER⁴</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Authorization ID of the owner of the index.</td>
</tr>
<tr>
<td>Column Name</td>
<td>Data Type</td>
<td>Nullable</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>---------------</td>
<td>----------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>LASTUSED</td>
<td>DATE</td>
<td></td>
<td>Date when the index was last used by any DML statement or used to enforce referential integrity constraints. The default value is '0001-01-01'. This value is updated asynchronously.</td>
</tr>
<tr>
<td>REMARKS</td>
<td>VARCHAR (254)</td>
<td>Y</td>
<td>User-provided comments, or the null value.</td>
</tr>
</tbody>
</table>

**Note:**
1. When using DMS table spaces, this statistic cannot be computed.
2. Prefetch statistics are not gathered during a LOAD...STATISTICS YES, or a CREATE INDEX...COLLECT STATISTICS operation, or when the database configuration parameter `seqdetect` is turned off.
3. `AVGPARTITION_CLUSTERRATIO`, `AVGPARTITION_CLUSTERFACTOR`, and `AVGPARTITION_PAGE_FETCH_PAIRS` measure the degree of clustering within a single data partition (local clustering). `CLUSTERRATIO`, `CLUSTERFACTOR`, and `PAGE_FETCH_PAIRS` measure the degree of clustering in the entire table (global clustering). Global clustering and local clustering values can diverge significantly if the table partitioning key is not a prefix of the index key, or when the table partitioning key and the index key are logically independent of each other.
4. The DEFINER column is included for backwards compatibility. See OWNER.
5. The XPATH, XRGN, and XVIP indexes are not recognized by any application programming interface that returns index metadata.
Each row represents an index exploitation rule.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FUNCID</td>
<td>INTEGER</td>
<td></td>
<td>Identifier for the function.</td>
</tr>
<tr>
<td>SPECID</td>
<td>SMALLINT</td>
<td></td>
<td>Number of the predicate specification.</td>
</tr>
<tr>
<td>IESCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the index extension.</td>
</tr>
<tr>
<td>IENAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the index extension.</td>
</tr>
<tr>
<td>RULEID</td>
<td>SMALLINT</td>
<td></td>
<td>Identifier for the exploitation rule.</td>
</tr>
<tr>
<td>SEARCHMETHODID</td>
<td>SMALLINT</td>
<td></td>
<td>Identifier for the search method in the specific index extension.</td>
</tr>
<tr>
<td>SEARCHKEY</td>
<td>VARCHAR (640)</td>
<td></td>
<td>Key used to exploit the index.</td>
</tr>
<tr>
<td>SEARCHARGUMENT</td>
<td>VARCHAR (2700)</td>
<td></td>
<td>Search arguments used to exploit the index.</td>
</tr>
<tr>
<td>EXACT</td>
<td>CHAR (1)</td>
<td></td>
<td>• N = Index lookup is not exact in terms of predicate evaluation</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = Index lookup is exact in terms of predicate evaluation</td>
</tr>
</tbody>
</table>
Each row represents a dependency of an index extension on some other object. The index extension depends on the object of type BTYPE of name BNAME, so a change to the object affects the index extension.

Table 125. SYSCAT.INDEXEXTENSIONDEP Catalog View

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IESCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the index extension.</td>
</tr>
<tr>
<td>IENAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the index extension.</td>
</tr>
<tr>
<td>BTYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>Type of object on which there is a dependency. Possible values are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• A = Table alias</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• B = Trigger</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• F = Routine</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• G = Global temporary table</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• H = Hierarchy table</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• K = Package</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• L = Detached table</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = Nickname</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• O = Privilege dependency on all subtables or subviews in a table or view hierarchy</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Q = Sequence</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• R = User-defined data type</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• S = Materialized query table</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• T = Table (not typed)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• U = Typed table</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• V = View (not typed)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• W = Typed view</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• X = Index extension</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Z = XSR object</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• q = Sequence alias</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• u = Module alias</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• v = Global variable</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• * = Anchored to the row of a base table</td>
</tr>
<tr>
<td>BSCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the object on which there is a dependency.</td>
</tr>
<tr>
<td>BMODULENAME</td>
<td>VARCHAR(128)</td>
<td>Y</td>
<td>Unqualified name of the module to which the object on which a dependency belongs. The null value if not a module object.</td>
</tr>
<tr>
<td>BNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the object on which there is a dependency. For routines (BTYPE = 'F'), this is the specific name.</td>
</tr>
<tr>
<td>BMODULEID</td>
<td>INTEGER</td>
<td>Y</td>
<td>Identifier for the module of the object on which there is a dependency.</td>
</tr>
<tr>
<td>TABAUTH</td>
<td>SMALLINT</td>
<td>Y</td>
<td>If BTYPE = ‘O’, ‘S’, ‘T’, ‘U’, ‘V’, ‘W’, or ‘v’, encodes the privileges on the table or view that are required by the dependent index extension; the null value otherwise.</td>
</tr>
</tbody>
</table>
Each row represents a search method. An index extension can contain more than one search method.

**Table 126. SYSCAT.INDEXEXTENSIONMETHODS Catalog View**

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>METHODNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Name of the search method.</td>
</tr>
<tr>
<td>METHODID</td>
<td>SMALLINT</td>
<td></td>
<td>Number of the method in the index extension.</td>
</tr>
<tr>
<td>IESCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the index extension on which this method is defined.</td>
</tr>
<tr>
<td>IENAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the index extension on which this method is defined.</td>
</tr>
<tr>
<td>RANGEFUNCSHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the range-through function.</td>
</tr>
<tr>
<td>RANGEFUNCNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the range-through function.</td>
</tr>
<tr>
<td>RANGESPECIFICNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Function-specific name of the range-through function.</td>
</tr>
<tr>
<td>FILTERFUNCSHEMA</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>Schema name of the filter function.</td>
</tr>
<tr>
<td>FILTERFUNCNAMESPACE</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>Unqualified name of the filter function.</td>
</tr>
<tr>
<td>FILTERSPECIFICNAME</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>Function-specific name of the filter function.</td>
</tr>
<tr>
<td>REMARKS</td>
<td>VARCHAR (254)</td>
<td>Y</td>
<td>User-provided comments, or the null value.</td>
</tr>
</tbody>
</table>
SYSCAT.INDEXEXTENSIONPARMS

Each row represents an index extension instance parameter or source key column.

Table 127. SYSCAT.INDEXEXTENSIONPARMS Catalog View

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IESCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the index extension.</td>
</tr>
<tr>
<td>IENAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the index extension.</td>
</tr>
<tr>
<td>ORDINAL</td>
<td>SMALLINT</td>
<td></td>
<td>Sequence number of the parameter or key column.</td>
</tr>
<tr>
<td>PARMNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Name of the parameter or key column.</td>
</tr>
<tr>
<td>TYPESCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the data type of the parameter or key column.</td>
</tr>
<tr>
<td>TYPENAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the data type of the parameter or key column.</td>
</tr>
<tr>
<td>LENGTH</td>
<td>INTEGER</td>
<td></td>
<td>Data type length of the parameter or key column.</td>
</tr>
<tr>
<td>SCALE</td>
<td>SMALLINT</td>
<td></td>
<td>Data type scale of the parameter or key column.</td>
</tr>
</tbody>
</table>
| PARMTYPE      | CHAR (1)       |          | v K = Source key column  
|               |                |          | v P = Index extension instance parameter               |
| CODEPAGE      | SMALLINT       |          | Code page of the index extension instance parameter; 0 if not a string type. |
| COLLATIONSHEMA| VARCHAR (128)  | Y        | For string types, the schema name of the collation for the parameter; the null value otherwise. |
| COLLATIONNAME | VARCHAR (128)  | Y        | For string types, the unqualified name of the collation for the parameter; the null value otherwise. |
### SYSCAT.INDEXEXTENSIONS

Each row represents an index extension.

**Table 128. SYSCAT.INDEXEXTENSIONS Catalog View**

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IESCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the index extension.</td>
</tr>
<tr>
<td>IENAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the index extension.</td>
</tr>
<tr>
<td>OWNER</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Authorization ID of the owner of the index extension.</td>
</tr>
<tr>
<td>OWNERTYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>• S = The owner is the system</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• U = The owner is an individual user</td>
</tr>
<tr>
<td>CREATE_TIME</td>
<td>TIMESTAMP</td>
<td></td>
<td>Time at which the index extension was defined.</td>
</tr>
<tr>
<td>KEYGENFUNCSHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the key generation function.</td>
</tr>
<tr>
<td>KEYGENFUNCNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the key generation function.</td>
</tr>
<tr>
<td>KEYGENSPECIFICNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Name of the key generation function instance (might be system-generated).</td>
</tr>
<tr>
<td>TEXT</td>
<td>CLOB (2M)</td>
<td></td>
<td>Full text of the CREATE INDEX EXTENSION statement.</td>
</tr>
<tr>
<td>DEFINER^</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Authorization ID of the owner of the index extension.</td>
</tr>
<tr>
<td>REMARKS</td>
<td>VARCHAR (254)</td>
<td>Y</td>
<td>User-provided comments, or the null value.</td>
</tr>
</tbody>
</table>

**Note:**

1. The DEFINER column is included for backwards compatibility. See OWNER.
Each row represents an index-specific option value.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INDSHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the index.</td>
</tr>
<tr>
<td>INDNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the index.</td>
</tr>
<tr>
<td>OPTION</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Name of the index option.</td>
</tr>
<tr>
<td>SETTING</td>
<td>VARCHAR (2048)</td>
<td></td>
<td>Value of the index option.</td>
</tr>
</tbody>
</table>
Each row represents a partitioned index piece located on one data partition. Note:
- The index partition statistics represent one database partition if the table is created on multiple database partitions.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INDSCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the index.</td>
</tr>
<tr>
<td>INDNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the index.</td>
</tr>
<tr>
<td>TABSCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the table or nickname on which the index is defined.</td>
</tr>
<tr>
<td>TABNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the table or nickname on which the index is defined.</td>
</tr>
<tr>
<td>IID</td>
<td>SMALLINT</td>
<td></td>
<td>Identifier for the index.</td>
</tr>
<tr>
<td>INDPARTITIONTSPACEID</td>
<td>INTEGER</td>
<td></td>
<td>Identifier for the index partition table space.</td>
</tr>
<tr>
<td>INDPARTITIONOBJECTID</td>
<td>INTEGER</td>
<td></td>
<td>Identifier for the index partition object.</td>
</tr>
<tr>
<td>DATAPARTITIONID</td>
<td>INTEGER</td>
<td></td>
<td>This corresponds to the DATAPARTITIONID found in the SYSCAT.DATAPARTITIONS view.</td>
</tr>
<tr>
<td>INDCARD</td>
<td>BIGINT</td>
<td></td>
<td>Cardinality of the index partition. This might be different from the cardinality of the corresponding data partition for partitioned indexes that do not have a one-to-one relationship between the data partition rows and the index entries.</td>
</tr>
<tr>
<td>NLEAF</td>
<td>BIGINT</td>
<td></td>
<td>Number of leaf pages in the index partition; -1 if statistics are not collected.</td>
</tr>
<tr>
<td>NUM_EMPTY_LEAFS</td>
<td>BIGINT</td>
<td></td>
<td>Total number of index leaf pages in the index partition that have all of their row identifiers (RIDs) or block identifiers (BIDs) marked deleted.</td>
</tr>
<tr>
<td>NUMRIDS</td>
<td>BIGINT</td>
<td></td>
<td>Total number of row identifiers (RIDs) or block identifiers (BIDs) in the index partition; -1 if not known.</td>
</tr>
<tr>
<td>NUMRIDS.DELETED</td>
<td>BIGINT</td>
<td></td>
<td>Total number of row identifiers (RIDs) or block identifiers (BIDs) in the index partition that are marked deleted, excluding those identifiers on leaf pages on which all the identifiers are marked deleted.</td>
</tr>
<tr>
<td>FULLKEYCARD</td>
<td>BIGINT</td>
<td></td>
<td>Number of distinct full-key values in the index partition; -1 if statistics are not collected.</td>
</tr>
<tr>
<td>NLEVELS</td>
<td>SMALLINT</td>
<td></td>
<td>Number of index levels in the index partition; -1 if statistics are not collected.</td>
</tr>
<tr>
<td>Column Name</td>
<td>Data Type</td>
<td>Nullable</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>-------------</td>
<td>----------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>CLUSTERRATIO</td>
<td>SMALLINT</td>
<td></td>
<td>Degree of data clustering with the index partition; -1 in either of the following situations:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Statistics are not collected</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Detailed index statistics are collected. In this situation, CLUSTERFACTOR will be used instead.</td>
</tr>
<tr>
<td>CLUSTERFACTOR</td>
<td>DOUBLE</td>
<td></td>
<td>Finer measurement of the degree of clustering; -1 if statistics are not collected.</td>
</tr>
<tr>
<td>FIRSTKEYCARD</td>
<td>BIGINT</td>
<td></td>
<td>Number of distinct first-key values; -1 if statistics are not collected.</td>
</tr>
<tr>
<td>FIRST2KEYCARD</td>
<td>BIGINT</td>
<td></td>
<td>Number of distinct keys using the first two columns of the index key; -1 if statistics are not collected, or if not applicable.</td>
</tr>
<tr>
<td>FIRST3KEYCARD</td>
<td>BIGINT</td>
<td></td>
<td>Number of distinct keys using the first three columns of the index key; -1 if statistics are not collected, or if not applicable.</td>
</tr>
<tr>
<td>FIRST4KEYCARD</td>
<td>BIGINT</td>
<td></td>
<td>Number of distinct keys using the first four columns of the index key; -1 if statistics are not collected, or if not applicable.</td>
</tr>
<tr>
<td>AVGLEAFKEYSIZE</td>
<td>INTEGER</td>
<td></td>
<td>Average index key size for keys on leaf pages in the index partition; -1 if statistics are not collected.</td>
</tr>
<tr>
<td>AVGNLEAFKEYSIZE</td>
<td>INTEGER</td>
<td></td>
<td>Average index key size for keys on non-leaf pages in the index partition; -1 if statistics are not collected.</td>
</tr>
<tr>
<td>PCTFREE</td>
<td>SMALLINT</td>
<td></td>
<td>Percentage of each index page to be reserved during the initial building of the index partition. This space is available for data insertions after the index partition has been built.</td>
</tr>
<tr>
<td>PAGE_FETCH_PAIRS</td>
<td>VARCHAR (520)</td>
<td></td>
<td>A list of pairs of integers, represented in character form. Each pair represents the number of pages in a hypothetical buffer, and the number of page fetches required to scan the data partition with this index using that hypothetical buffer. Zero-length string if not data is available.</td>
</tr>
<tr>
<td>SEQUENTIAL_PAGES</td>
<td>BIGINT</td>
<td></td>
<td>Number of leaf pages located on disk in index key order with few or no large gaps between them; -1 if statistics are not collected.</td>
</tr>
<tr>
<td>DENSITY</td>
<td>INTEGER</td>
<td></td>
<td>Ratio of SEQUENTIAL_PAGES to number of pages in the range of pages occupied by the index partition, expressed as a percent (integer between 0 and 100); -1 if statistics are not collected.</td>
</tr>
<tr>
<td>Column Name</td>
<td>Data Type</td>
<td>Nullable</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>--------------</td>
<td>----------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>AVERAGE_SEQUENCE_GAP</td>
<td>DOUBLE</td>
<td></td>
<td>Gap between index page sequences within the index partition. Detected through a scan of index leaf pages, each gap represents the average number of index pages that must be randomly fetched between sequences of index pages; -1 if not known.</td>
</tr>
<tr>
<td>AVERAGE_SEQUENCE_FETCH_GAP</td>
<td>DOUBLE</td>
<td></td>
<td>Gap between table page sequences when fetching using the index partition. Detected through a scan of index leaf pages, each gap represents the average number of data partition pages that must be randomly fetched between sequences of data partition pages; -1 if not known.</td>
</tr>
<tr>
<td>AVERAGE_SEQUENCE_PAGES</td>
<td>DOUBLE</td>
<td></td>
<td>Average number of index pages that are accessible in sequence (that is, the number of index pages that the prefetchers would detect as being in sequence); -1 if not known.</td>
</tr>
<tr>
<td>AVERAGE_SEQUENCE_FETCH_PAGES</td>
<td>DOUBLE</td>
<td></td>
<td>Average number of data partition pages that are accessible in sequence (that is, the number of data partition pages that the prefetchers would detect as being in sequence) when fetching using the index; -1 if not known.</td>
</tr>
<tr>
<td>AVERAGE_RANDOM_PAGES</td>
<td>DOUBLE</td>
<td></td>
<td>Average number of random data partition pages between sequential page accesses; -1 if not known.</td>
</tr>
<tr>
<td>AVERAGE_RANDOM_FETCH_PAGES</td>
<td>DOUBLE</td>
<td></td>
<td>Average number of random data partition pages between sequential page accesses when fetching using the index partition; -1 if not known.</td>
</tr>
<tr>
<td>STATS_TIME</td>
<td>TIMESTAMP</td>
<td>Y</td>
<td>Last time that any change was made to the recorded statistics for this index partition. The null value if no statistics are available.</td>
</tr>
<tr>
<td>COMPRESSION</td>
<td>CHAR (1)</td>
<td></td>
<td>Specifies whether index compression is activated</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = Not activated</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = Activated</td>
</tr>
<tr>
<td>PCTPAGESAVED</td>
<td>SMALLINT</td>
<td></td>
<td>Approximate percentage of pages saved in the index as a result of index compression. -1 if statistics are not collected.</td>
</tr>
</tbody>
</table>
SYSCAT.INDEXXMLPATTERNS

Each row represents a pattern clause in an index over an XML column.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INDSHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the logical index.</td>
</tr>
<tr>
<td>INDNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the logical index.</td>
</tr>
<tr>
<td>PINDNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the physical index.</td>
</tr>
<tr>
<td>PINDID</td>
<td>SMALLINT</td>
<td></td>
<td>Identifier for the physical index.</td>
</tr>
<tr>
<td>TYPEMODEL</td>
<td>CHAR (1)</td>
<td></td>
<td>• Q = SQL DATA TYPE (Ignore invalid values)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• R = SQL DATA TYPE (Reject invalid values)</td>
</tr>
<tr>
<td>DATATYPE</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Name of the data type.</td>
</tr>
<tr>
<td>HASHED</td>
<td>CHAR (1)</td>
<td></td>
<td>Indicates whether or not the value is hashed.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = Not hashed</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = Hashed</td>
</tr>
<tr>
<td>LENGTH</td>
<td>SMALLINT</td>
<td></td>
<td>VARCHAR (n) length; 0 otherwise.</td>
</tr>
<tr>
<td>PATTERNID</td>
<td>SMALLINT</td>
<td></td>
<td>Identifier for the pattern.</td>
</tr>
<tr>
<td>PATTERN</td>
<td>CLOB (2M)</td>
<td>Y</td>
<td>Definition of the pattern.</td>
</tr>
</tbody>
</table>

**Note:**

1. When indexes over XML columns are created, logical indexes that utilize XML pattern information are created, resulting in the creation of physical B-tree indexes with DB2-generated key columns to support the logical indexes. A physical index is created to support the data type that is specified in the xmltype-clause of the CREATE INDEX statement.
Each row represents an invalid object.

### Table 132. SYSCAT.INVALIDOBJECTS Catalog View

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OBJECTSCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the object being created or revalidated.</td>
</tr>
<tr>
<td>OBJECTMODULENAME</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>Unqualified name of the module to which the object belongs. The null value if the object does not belong to a module.</td>
</tr>
<tr>
<td>OBJECTNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the object being created or revalidated. For routines (OBJECTTYPE = 'F'), this is the specific name.</td>
</tr>
<tr>
<td>ROUTINENAME</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>Unqualified name of the routine.</td>
</tr>
<tr>
<td>OBJECTTYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>Type of the object being created or revalidated. Possible values are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- B = Trigger</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- F = Routine</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- R = User-defined data type</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- V = View</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- v = Global variable</td>
</tr>
<tr>
<td>SQLCODE</td>
<td>INTEGER</td>
<td>Y</td>
<td>SQLCODE returned in CREATE with errors or revalidation. The null value if the object has never been revalidated.</td>
</tr>
<tr>
<td>SQLSTATE</td>
<td>CHAR (5)</td>
<td>Y</td>
<td>SQLSTATE returned in CREATE with errors or revalidation. The null value if the object has never been revalidated.</td>
</tr>
<tr>
<td>ERRORMESSAGE</td>
<td>VARCHAR(70)</td>
<td>Y</td>
<td>Short text for the message associated with SQLCODE. The null value if the object has never been revalidated.</td>
</tr>
<tr>
<td>LINENUMBER</td>
<td>INTEGER</td>
<td>Y</td>
<td>Line number where the error occurred in compiled objects. The null value if the object is not a compiled object.</td>
</tr>
<tr>
<td>INVALIDATE_TIME</td>
<td>TIMESTAMP</td>
<td></td>
<td>Time at which the object was last invalidated.</td>
</tr>
<tr>
<td>LAST_REGEN_TIME</td>
<td>TIMESTAMP</td>
<td>Y</td>
<td>Time at which the object was last revalidated. The null value if the object has never been revalidated.</td>
</tr>
</tbody>
</table>
Each row represents a column that participates in a key defined by a unique, primary key, or foreign key constraint.

Table 133. SYSCAT.KEYCOLUSE Catalog View

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONSTNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Name of the constraint.</td>
</tr>
<tr>
<td>TABSCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the table containing the column.</td>
</tr>
<tr>
<td>TABNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the table containing the column.</td>
</tr>
<tr>
<td>COLNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Name of the column.</td>
</tr>
<tr>
<td>COLSEQ</td>
<td>SMALLINT</td>
<td></td>
<td>Numeric position of the column in the key for the constraint (initial position is 1). If a constraint uses an existing index, this value is the numeric position of the column in the index.</td>
</tr>
</tbody>
</table>
SYSCAT.MODULEAUTH

Each row represents a user, group, or role that has been granted a privilege on a module.

Table 134. SYSCAT.MODULEAUTH Catalog View

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRANTOR</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Grantor of a privilege</td>
</tr>
<tr>
<td>GRANTORTYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>• S = Grantor is the system</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• U = Grantor is an individual user</td>
</tr>
<tr>
<td>GRANTEE</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Holder of a privilege</td>
</tr>
<tr>
<td>GRANTEEETYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>• G = Grantee is a group</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• R = Grantee is a role</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• U = Grantee is an individual user</td>
</tr>
<tr>
<td>MODULEID</td>
<td>INTEGER</td>
<td></td>
<td>Identifier for the module to which this privilege applies.</td>
</tr>
<tr>
<td>MODULESCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the module to which this privilege applies.</td>
</tr>
<tr>
<td>MODULENAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the module to which this privilege applies.</td>
</tr>
<tr>
<td>EXECUTEAUTH</td>
<td>CHAR (1)</td>
<td></td>
<td>Privilege to execute objects in the identified module.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• G = Held and grantable</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = Not held</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = Held</td>
</tr>
</tbody>
</table>
SYSCAT.MODULEOBJECTS

Each row represents a function, procedure, global variable, condition, or user-defined type that belongs to a module.

Table 135. SYSCAT.MODULEOBJECTS Catalog View

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OBJECTSCHEMA</td>
<td>VARCHAR(128)</td>
<td>N</td>
<td>Schema name of the module.</td>
</tr>
<tr>
<td>OBJECTMODULENAME</td>
<td>VARCHAR(128)</td>
<td>N</td>
<td>Unqualified name of the module to which the object belongs.</td>
</tr>
<tr>
<td>OBJECTNAME</td>
<td>VARCHAR(128)</td>
<td>N</td>
<td>Unqualified name of the object.</td>
</tr>
<tr>
<td>OBJECTTYPE</td>
<td>VARCHAR(9)</td>
<td>N</td>
<td>• CONDITION = The object is a condition</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• FUNCTION = The object is a function</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• PROCEDURE = The object is a procedure</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• TYPE = The object is a data type</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• VARIABLE = The object is a variable</td>
</tr>
<tr>
<td>PUBLISHED</td>
<td>CHAR(1)</td>
<td>N</td>
<td>Indicates whether the object can be referenced outside its module.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = The object is not published</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = The object is published</td>
</tr>
<tr>
<td>SPECIFICNAME</td>
<td>VARCHAR(128)</td>
<td>N</td>
<td>Routine specific name if OBJECTTYPE is 'FUNCTION', 'METHOD' or 'PROCEDURE'; the null value otherwise.</td>
</tr>
<tr>
<td>USERDEFINED</td>
<td>CHAR(1)</td>
<td>N</td>
<td>Indicates whether the object is generated by the system or defined by a user.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = The object is system generated</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = The object is defined by a user</td>
</tr>
</tbody>
</table>
Table 136. SYSCAT.MODULES Catalog View

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MODULESCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the module.</td>
</tr>
<tr>
<td>MODULENAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the module.</td>
</tr>
<tr>
<td>MODULEID</td>
<td>INTEGER</td>
<td></td>
<td>Identifier for the module.</td>
</tr>
<tr>
<td>DIALECT</td>
<td>VARCHAR(10)</td>
<td></td>
<td>The source dialect of the SQL module. Possible values are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• DB2 SQL PL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• PL/SQL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Blank = Not applicable for an alias</td>
</tr>
<tr>
<td>OWNER</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Authorization ID of the owner of the module.</td>
</tr>
<tr>
<td>OWNERTYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>• S = The owner is the system</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• U = The owner is an individual user</td>
</tr>
<tr>
<td>MODULETYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>Type of module.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• A = Alias</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• M = Module</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• P = PL/SQL package</td>
</tr>
<tr>
<td>BASE_MODULESCHEMA</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>If MODULETYPE is ‘A’, contains the schema name of the module or alias that is referenced by this alias; the null value otherwise.</td>
</tr>
<tr>
<td>BASE_MODULENAME</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>If MODULETYPE is ‘A’, contains the unqualified name of the module or alias that is referenced by this alias; the null value otherwise.</td>
</tr>
<tr>
<td>CREATE_TIME</td>
<td>TIMESTAMP</td>
<td></td>
<td>Time at which the module was created.</td>
</tr>
<tr>
<td>REMARKS</td>
<td>VARCHAR (254)</td>
<td>Y</td>
<td>User-provided comments, or the null value.</td>
</tr>
</tbody>
</table>
Each row represents the mapping between a "logical" object (typed table or view and its columns and indexes, including inherited columns) and the corresponding “implementation” object (hierarchy table or hierarchy view and its columns and indexes) that implements the logical object.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>• C = Column</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• I = Index</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• U = Typed table</td>
</tr>
<tr>
<td>LOGICAL_SCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the logical object.</td>
</tr>
<tr>
<td>LOGICAL_NAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the logical object.</td>
</tr>
<tr>
<td>LOGICAL_COLNAME</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>Name of the logical column if TYPE = ‘C’; null value otherwise.</td>
</tr>
<tr>
<td>IMPL_SCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the implementation object that implements the logical object.</td>
</tr>
<tr>
<td>IMPL_NAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the implementation object that implements the logical object.</td>
</tr>
<tr>
<td>IMPL_COLNAME</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>Name of the implementation column if TYPE = ‘C’; null value otherwise.</td>
</tr>
</tbody>
</table>
### SYSCAT.NICKNAMES

Each row represents a nickname.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TABSCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the nickname.</td>
</tr>
<tr>
<td>TABNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the nickname.</td>
</tr>
<tr>
<td>OWNER</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Authorization ID of the owner of the table, view, alias, or nickname.</td>
</tr>
</tbody>
</table>
| OWNERTYPE   | CHAR (1) | | - S = The owner is the system  
- U = The owner is an individual user |
| STATUS      | CHAR (1) | | - C = Set integrity pending  
- N = Normal  
- X = Inoperative |
<p>| CREATE_TIME | TIMESTAMP | | Time at which the object was created. |
| STATS_TIME  | TIMESTAMP | Y | Time at which any change was last made to recorded statistics for this object. The null value if statistics are not collected. |
| COLCOUNT    | SMALLINT | | Number of columns, including inherited columns (if any). |
| TABLEID     | SMALLINT | | Internal logical object identifier. |
| TBSPACEID   | SMALLINT | | Internal logical identifier for the primary table space for this object. |
| CARD        | BIGINT | | Total number of rows in the table; -1 if statistics are not collected. |
| NPAGES      | BIGINT | | Total number of pages on which the rows of the nickname exist; -1 if statistics are not gathered. |
| FPAGES      | BIGINT | | Total number of pages; -1 if statistics are not gathered. |
| OVERFLOW    | BIGINT | | Total number of overflow records; -1 if statistics are not gathered. |
| PARENTS     | SMALLINT | Y | Number of parent tables for this object; that is, the number of referential constraints in which this object is a dependent. |
| CHILDREN    | SMALLINT | Y | Number of dependent tables for this object; that is, the number of referential constraints in which this object is a parent. |
| SELFREFS    | SMALLINT | Y | Number of self-referencing referential constraints for this object; that is, the number of referential constraints in which this object is both a parent and a dependent. |
| KEYCOLUMNS  | SMALLINT | Y | Number of columns in the primary key. |
| KEYINDEXID  | SMALLINT | Y | Index identifier for the primary key index; 0 or the null value if there is no primary key. |</p>
<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>KEYUNIQUE</td>
<td>SMALLINT</td>
<td></td>
<td>Number of unique key constraints (other than the primary key constraint) defined on this object.</td>
</tr>
<tr>
<td>CHECKCOUNT</td>
<td>SMALLINT</td>
<td></td>
<td>Number of check constraints defined on this object.</td>
</tr>
</tbody>
</table>
| DATACAPTURE  | CHAR (1)  |          | • L = Nickname participates in data replication, including replication of LONG VARCHAR and LONG VARGRAPHIC columns  
|              |           |          | • N = Nickname does not participate in data replication                       |
|              |           |          | • Y = Nickname participates in data replication                               |
| CONST_CHECKED| CHAR (32) |          | • Byte 1 represents foreign key constraint.                                 |
|              |           |          | • Byte 2 represents check constraint.                                        |
|              |           |          | • Byte 5 represents materialized query table.                               |
|              |           |          | • Byte 6 represents generated column.                                        |
|              |           |          | • Byte 7 represents staging table.                                            |
|              |           |          | • Byte 8 represents data partitioning constraint.                           |
|              |           |          | • Other bytes are reserved for future use.                                  |
| Partitions   | CHAR (1)  |          | Reserved for future use.                                                    |
| STATISTICS_PROFILE | CLOB (10M) | Y | RUNSTATS command used to register a statistical profile for the object. |
| ACCESS_MODE  | CHAR (1)  |          | Access restriction state of the object. These states only apply to objects that are in set integrity pending state or to objects that were processed by a SET INTEGRITY statement. Possible values are:  
|              |           |          | • D = No data movement                                                      |
|              |           |          | • F = Full access                                                           |
|              |           |          | • N = No access                                                             |
| CODEPAGE     | SMALLINT  |          | Code page of the object. This is the default code page used for all character columns, triggers, check constraints, and expression-generated columns. |
### SYSCAT.NICKNAMES

**Table 138. SYSCAT.NICKNAMES Catalog View (continued)**

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>REMOTE_TABLE</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>Unqualified name of the specific data source object (such as a table or a view) for which the nickname was created.</td>
</tr>
<tr>
<td>REMOTE_SCHEMA</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>Schema name of the specific data source object (such as a table or a view) for which the nickname was created.</td>
</tr>
<tr>
<td>SERVERNAME</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>Name of the data source that contains the table or view for which the nickname was created.</td>
</tr>
<tr>
<td>REMOTE_TYPE</td>
<td>CHAR (1)</td>
<td>Y</td>
<td>Type of object at the data source.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• A = Alias</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = Nickname</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• S = Materialized query table</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• T = Table (untyped)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• V = View (untyped)</td>
</tr>
<tr>
<td>CACHINGALLOWED</td>
<td>VARCHAR (1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = Caching is not allowed</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = Caching is allowed</td>
</tr>
<tr>
<td>DEFINER¹</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Authorization ID of the owner of the table, view, alias, or nickname.</td>
</tr>
<tr>
<td>REMARKS</td>
<td>VARCHAR (254)</td>
<td>Y</td>
<td>User-provided comments, or the null value.</td>
</tr>
</tbody>
</table>

**Note:**

1. The DEFINER column is included for backwards compatibility. See OWNER.
Each row represents a user, group, or role that has been granted one or more privileges on a package.

**Table 139. SYSCAT.PACKAGEAUTH Catalog View**

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRANTOR</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Grantor of the privilege.</td>
</tr>
<tr>
<td>GRANTORTYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>• S = Grantor is the system</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• U = Grantor is an individual user</td>
</tr>
<tr>
<td>GRANTEE</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Holder of the privilege.</td>
</tr>
<tr>
<td>GRANTEEETYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>• G = Grantee is a group</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• R = Grantee is a role</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• U = Grantee is an individual user</td>
</tr>
<tr>
<td>PKGSHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the package.</td>
</tr>
<tr>
<td>PKGNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the package.</td>
</tr>
<tr>
<td>CONTROLAUTH</td>
<td>CHAR (1)</td>
<td></td>
<td>CONTROL privilege.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = Not held</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = Held</td>
</tr>
<tr>
<td>BINDAUTH</td>
<td>CHAR (1)</td>
<td></td>
<td>Privilege to bind the package.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• G = Held and grantable</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = Not held</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = Held</td>
</tr>
<tr>
<td>EXECUTEAUTH</td>
<td>CHAR (1)</td>
<td></td>
<td>Privilege to execute the package.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• G = Held and grantable</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = Not held</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = Held</td>
</tr>
</tbody>
</table>
Each row represents a dependency of a package on some other object. The package depends on the object of type BTYPE of name BNAME, so a change to the object affects the package.

Table 140. SYSCAT.PACKAGEDEP Catalog View

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PKGSHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the package.</td>
</tr>
<tr>
<td>PKGNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the package.</td>
</tr>
<tr>
<td>BINDER</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Binder of the package.</td>
</tr>
<tr>
<td>BINDERTYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>• U = Binder is an individual user</td>
</tr>
<tr>
<td>BTYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>Type of object on which there is a dependency. Possible values are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• A = Table alias</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• B = Trigger</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• D = Server definition</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• F = Routine</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• G = User temporary table</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• I = Index</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• M = Function mapping</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = Nickname</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• O = Privilege dependency on all subtables or subviews in a table or view hierarchy</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• P = Page size</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Q = Sequence object</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• R = User-defined data type</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• S = Materialized query table</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• T = Table (untyped)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• U = Typed table</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• V = View (untyped)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• W = Typed view</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Z = XSR object</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• m = Module</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• n = Database partition group</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• q = Sequence alias</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• u = Module alias</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• v = Global variable</td>
</tr>
<tr>
<td>BSCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of an object on which the package depends.</td>
</tr>
<tr>
<td>BMODULENAME</td>
<td>VARCHAR(128)</td>
<td>Y</td>
<td>Unqualified name of the module to which the object on which a dependency belongs. The null value if not a module object.</td>
</tr>
<tr>
<td>BNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of an object on which the package depends.</td>
</tr>
<tr>
<td>BMODULEID</td>
<td>INTEGER</td>
<td>Y</td>
<td>Identifier for the module of the object on which there is a dependency.</td>
</tr>
</tbody>
</table>
### SYSCAT.PACKAGEDEP

Table 140. SYSCAT.PACKAGEDEP Catalog View (continued)

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TABAUTH</td>
<td>SMALLINT</td>
<td>Y</td>
<td>If BTYPE is 'O', 'S', 'T', 'U', 'V', or 'W', encodes the privileges that are required by this package (SELECT, INSERT, UPDATE, or DELETE). Bit vector is defined in SQL.H.</td>
</tr>
<tr>
<td>VARAUTH</td>
<td>SMALLINT</td>
<td>Y</td>
<td>If BTYPE is 'v', encodes the privileges that are required by this package (READ or WRITE). Bit vector is defined in SQL.H.</td>
</tr>
<tr>
<td>UNIQUE_ID</td>
<td>CHAR (8) FOR BIT DATA</td>
<td></td>
<td>Identifier for a specific package when multiple packages having the same name exist.</td>
</tr>
<tr>
<td>PKGVERSION</td>
<td>VARCHAR (64)</td>
<td>Y</td>
<td>Version identifier for the package.</td>
</tr>
</tbody>
</table>

**Note:**

1. If a function instance with dependencies is dropped, the package is put into an “inoperative” state, and it must be explicitly rebound. If any other object with dependencies is dropped, the package is put into an “invalid” state, and the system will attempt to rebind the package automatically when it is first referenced.
Each row represents a package that has been created by binding an application program.

### Table 141. SYSCAT.PACKAGES Catalog View

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PKGSHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the package.</td>
</tr>
<tr>
<td>PKGNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the package.</td>
</tr>
<tr>
<td>BOUNDBY</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Authorization ID of the binder and owner of the package.</td>
</tr>
<tr>
<td>BOUNDBYTYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>• U = The binder and owner is an individual user</td>
</tr>
<tr>
<td>OWNER</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Authorization ID of the binder and owner of the package.</td>
</tr>
<tr>
<td>OWNERTYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>• U = The binder and owner is an individual user</td>
</tr>
<tr>
<td>DEFAULT_SCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Default schema name used for unqualified names in static SQL statements.</td>
</tr>
<tr>
<td>VALID</td>
<td>CHAR (1)</td>
<td></td>
<td>• N = Needs rebinding</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• V = Validate at run time</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• X = Package is inoperative because some function instance on which it</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>depends has been dropped; explicit rebinding is needed</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = Valid</td>
</tr>
<tr>
<td>UNIQUE_ID</td>
<td>CHAR (8) FOR BIT DATA</td>
<td></td>
<td>Identifier for a specific package when multiple packages having the same</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>name exist.</td>
</tr>
<tr>
<td>TOTAL_SECT</td>
<td>SMALLINT</td>
<td></td>
<td>Number of sections in the package.</td>
</tr>
<tr>
<td>FORMAT</td>
<td>CHAR (1)</td>
<td></td>
<td>Date and time format associated with the package.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• 0 = Format associated with the territory code of the client</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• 1 = USA</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• 2 = EUR</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• 3 = ISO</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• 4 = JIS</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• 5 = LOCAL</td>
</tr>
<tr>
<td>ISOLATION</td>
<td>CHAR (2)</td>
<td>Y</td>
<td>Isolation level.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• CS = Cursor Stability</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• RR = Repeatable Read</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• RS = Read Stability</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• UR = Uncommitted Read</td>
</tr>
<tr>
<td>CONCURRENTACCESSRESOLUTION</td>
<td>CHAR (1)</td>
<td>Y</td>
<td>The value of the CONCURRENTACCESSRESOLUTION bind option:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• U = USE CURRENTLY COMMITTED</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• W = WAIT FOR OUTCOME</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Blank = Not specified</td>
</tr>
<tr>
<td>BLOCKING</td>
<td>CHAR (1)</td>
<td>Y</td>
<td>Cursor blocking option.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• B = Block all cursors</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = No blocking</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• U = Block unambiguous cursors</td>
</tr>
<tr>
<td>INSERT_BUF</td>
<td>CHAR (1)</td>
<td></td>
<td>Setting of the INSERT bind option (applies to partitioned database systems).</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = Inserts are not buffered</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = Inserts are buffered at the coordinator database partition to minimize</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>traffic among database partitions</td>
</tr>
<tr>
<td>LANG_LEVEL</td>
<td>CHAR (1)</td>
<td>Y</td>
<td>Setting of the LANG_LEVEL bind option.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• 0 = SAA1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• 1 = MIA</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• 2 = SQL92E</td>
</tr>
<tr>
<td>FUNC_PATH</td>
<td>CLOB (2K)</td>
<td></td>
<td>SQL path in effect when the package was bound.</td>
</tr>
</tbody>
</table>
### SYSCAT.PACKAGES

**Table 141. SYSCAT.PACKAGES Catalog View (continued)**

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>QUERYOPT</td>
<td>INTEGER</td>
<td></td>
<td>Optimization class under which this package was bound. Used for rebind operations.</td>
</tr>
<tr>
<td>EXPLAIN_LEVEL</td>
<td>CHAR (1)</td>
<td></td>
<td>Indicates whether Explain was requested using the EXPLAIN or EXPLSNAP bind option.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>P = Package selection level</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Blank = No Explain requested</td>
</tr>
<tr>
<td>EXPLAIN_MODE</td>
<td>CHAR (1)</td>
<td></td>
<td>Value of the EXPLAIN bind option.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>A = ALL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>N = No</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>R = REOPT</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Y = Yes</td>
</tr>
<tr>
<td>EXPLAIN_SNAPSHOT</td>
<td>CHAR (1)</td>
<td></td>
<td>Value of the EXPLSNAP bind option.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>A = ALL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>N = No</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>R = REOPT</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Y = Yes</td>
</tr>
<tr>
<td>SQLWARN</td>
<td>CHAR (1)</td>
<td></td>
<td>Indicates whether or not positive SQLCODEs resulting from dynamic SQL statements are returned to the application.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>N = No, they are suppressed</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Y = Yes</td>
</tr>
<tr>
<td>SQLMATHWARN</td>
<td>CHAR (1)</td>
<td></td>
<td>Value of the dft_sqlmathwarn database configuration parameter at bind time. Indicates whether arithmetic and retrieval conversion errors return warnings and null values (indicator -2), allowing query processing to continue whenever possible.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>N = No, errors are returned</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Y = Yes</td>
</tr>
<tr>
<td>CREATE_TIME</td>
<td>TIMESTAMP</td>
<td></td>
<td>Time at which the package was first bound.</td>
</tr>
<tr>
<td>EXPLICIT_BIND_TIME</td>
<td>TIMESTAMP</td>
<td></td>
<td>Time at which this package was last changed by:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>BIND</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>REBIND (explicit)</td>
</tr>
<tr>
<td>LAST_BIND_TIME</td>
<td>TIMESTAMP</td>
<td></td>
<td>Time at which the package was last changed by:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>BIND</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>REBIND (explicit)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>REBIND (implicit)</td>
</tr>
<tr>
<td>ALTER_TIME</td>
<td>TIMESTAMP</td>
<td></td>
<td>Time at which this package was last changed by:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>BIND</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>REBIND (explicit)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>REBIND (implicit)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ALTER PACKAGE</td>
</tr>
<tr>
<td>CODEPAGE</td>
<td>SMALLINT</td>
<td></td>
<td>Application code page at bind time; -1 if not known.</td>
</tr>
<tr>
<td>COLLATIONSHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the collation for the package.</td>
</tr>
<tr>
<td>COLLATIONNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the collation for the package.</td>
</tr>
<tr>
<td>COLLATIONSCHEMA_ORDERBY</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the collation for ORDER BY clauses in the package.</td>
</tr>
<tr>
<td>COLLATIONNAME_ORDERBY</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the collation for ORDER BY clauses in the package.</td>
</tr>
<tr>
<td>DEGREE</td>
<td>CHAR (5)</td>
<td></td>
<td>Degree of intra-partition parallelism that was specified when the package was bound.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1 = No parallelism</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2-32767 = User-specified limit</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ANY = Degree determined by the system (no limit specified)</td>
</tr>
<tr>
<td>Column Name</td>
<td>Data Type</td>
<td>Nullable</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>---------------</td>
<td>----------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>MULTINODE_PLANS</td>
<td>CHAR (1)</td>
<td></td>
<td>• N = Package was not bound in a partitioned database environment</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = Package was bound in a partitioned database environment</td>
</tr>
<tr>
<td>INTRA_PARALLEL</td>
<td>CHAR (1)</td>
<td></td>
<td>Use of intra-partition parallelism by static SQL statements within the package.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• F = One or more static SQL statements in this package can use intra-partition parallelism; this parallelism has been disabled for use on a system that is not configured for intra-partition parallelism</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = No static SQL statement uses intra-partition parallelism</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = One or more static SQL statements in the package use intra-partition parallelism</td>
</tr>
<tr>
<td>VALIDATE</td>
<td>CHAR (1)</td>
<td></td>
<td>Indicates whether validation can be deferred until run time.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• B = All checking must be performed at bind time</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• R = Validation of tables, views, and privileges that do not exist at bind time is performed at run time</td>
</tr>
<tr>
<td>DYNAMICRULES</td>
<td>CHAR (1)</td>
<td></td>
<td>• B = BIND; dynamic SQL statements are executed with DYNAMICRULES BIND behavior</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• D = DEFINERBIND; when the package is run within a routine context, dynamic SQL statements in the package are executed with DEFINE behavior; when the package is not run within a routine context, dynamic SQL statements in the package are executed with BIND behavior</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• E = DEFINERRUN; when the package is run within a routine context, dynamic SQL statements in the package are executed with DEFINE behavior; when the package is not run within a routine context, dynamic SQL statements in the package are executed with RUN behavior</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• H = INVOKEBIND; when the package is run within a routine context, dynamic SQL statements in the package are executed with INVOKE behavior; when the package is not run within a routine context, dynamic SQL statements in the package are executed with BIND behavior</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• I = INVOKERUN; when the package is run within a routine context, dynamic SQL statements in the package are executed with INVOKE behavior; when the package is not run within a routine context, dynamic SQL statements in the package are executed with RUN behavior</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• R = RUN; dynamic SQL statements are executed with RUN behavior; this is the default</td>
</tr>
<tr>
<td>SQLERROR</td>
<td>CHAR (1)</td>
<td></td>
<td>SQLERROR option on the most recent subcommand that bound or rebound the package.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• C = CONTINUE; creates a package, even if errors occur while binding SQL statements</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = NOPACKAGE; does not create a package or a bind file if an error occurs</td>
</tr>
<tr>
<td>REFRESHAGE</td>
<td>DECIMAL (20,6)</td>
<td></td>
<td>Timestamp duration indicating the maximum length of time between execution of a REFRESH TABLE statement for a materialized query table (MQT) and when that MQT is used in place of a base table.</td>
</tr>
<tr>
<td>FEDERATED</td>
<td>CHAR (1)</td>
<td></td>
<td>• N = FEDERATED bind or prep option is turned off</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• U = FEDERATED bind or prep option was not specified</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = FEDERATED bind or prep option is turned on</td>
</tr>
<tr>
<td>TRANSFORMGROUP</td>
<td>VARCHAR (1024)</td>
<td>Y</td>
<td>Value of the TRANSFORM GROUP bind option; the null value if a transform group is not specified.</td>
</tr>
</tbody>
</table>
### Table 141. SYSCAT.PACKAGES Catalog View (continued)

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>REOPTVAR</td>
<td>CHAR (1)</td>
<td></td>
<td>Indicates whether the access path is determined again at execution time using input variable values.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• <strong>A</strong> = Access path is reoptimized for every OPEN or EXECUTE request</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• <strong>N</strong> = Access path is determined at bind time</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• <strong>O</strong> = Access path is reoptimized only at the first OPEN or EXECUTE request; it is subsequently cached</td>
</tr>
<tr>
<td>OS_PTR_SIZE</td>
<td>INTEGER</td>
<td></td>
<td>Word size for the platform on which the package was created.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• <strong>32</strong> = Package is a 32-bit package</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• <strong>64</strong> = Package is a 64-bit package</td>
</tr>
<tr>
<td>PKGVERSION</td>
<td>VARCHAR (64)</td>
<td></td>
<td>Version identifier for the package.</td>
</tr>
<tr>
<td>STATICREADONLY</td>
<td>CHAR (1)</td>
<td></td>
<td>Indicates whether or not static cursors will be treated as READ ONLY.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• <strong>N</strong> = Static cursors take on the attributes that would normally be generated for the given statement text and the setting of the LANGLEVEL precompile option</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• <strong>Y</strong> = Any static cursor that does not contain the FOR UPDATE or the FOR READ ONLY clause is considered READ ONLY</td>
</tr>
<tr>
<td>FEDERATED_ASYNCHRONY</td>
<td>INTEGER</td>
<td></td>
<td>Indicates the limit on asynchrony (the number of ATQs in the plan) as a bind option when the package was bound.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• <strong>0</strong> = No asynchrony</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• <strong>n</strong> = User-specified limit (32,767 maximum)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• <strong>-1</strong> = Degree of asynchrony determined by the system</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• <strong>-2</strong> = Degree of asynchrony not specified</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>For a non-federated system, the value is 0.</td>
</tr>
<tr>
<td>ANONBLOCK</td>
<td>CHAR (1)</td>
<td></td>
<td>Indicates whether the package is associated with an anonymous block.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• <strong>N</strong> = The package is not associated with an anonymous block</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• <strong>Y</strong> = The package is associated with an anonymous block</td>
</tr>
<tr>
<td>OPTPROFILESCHEMA</td>
<td>VARCHAR (128)</td>
<td><strong>Y</strong></td>
<td>Value of the optimization profile schema specified as part of the OPTPROFILE bind option.</td>
</tr>
<tr>
<td>OPTPROFILENAME</td>
<td>VARCHAR (128)</td>
<td><strong>Y</strong></td>
<td>Value of the optimization profile name specified as part of the OPTPROFILE bind option.</td>
</tr>
<tr>
<td>PKGID</td>
<td>BIGINT</td>
<td></td>
<td>Identifier for the package.</td>
</tr>
<tr>
<td>DBPARTITIONNUM</td>
<td>SMALLINT</td>
<td></td>
<td>Number of the database partition where the package was bound.</td>
</tr>
<tr>
<td>DEFINER²</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Authorization ID of the binder and owner of the package.</td>
</tr>
<tr>
<td>PKG_CREATE_TIME³</td>
<td>TIMESTAMP⁵</td>
<td></td>
<td>Time at which the package was first bound.</td>
</tr>
<tr>
<td>APREUSE</td>
<td>CHAR (1)</td>
<td></td>
<td>• <strong>N</strong> = The query compiler will not attempt to reuse access plans</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• <strong>Y</strong> = The access plans in this package should be reused, meaning that at rebind time the query compiler will attempt to choose plans like the ones currently in the package</td>
</tr>
<tr>
<td>EXTENDEDINDICATOR</td>
<td>CHAR (1)</td>
<td></td>
<td>• <strong>N</strong> = Extended indicator variable values are not recognized</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• <strong>Y</strong> = Extended indicator variable values are recognized</td>
</tr>
<tr>
<td>LASTUSED</td>
<td>DATE</td>
<td></td>
<td>Date when any statement in the package was last executed. This column is not updated for a package associated with an anonymous block.</td>
</tr>
<tr>
<td>REMARKS</td>
<td>VARCHAR (254)</td>
<td><strong>Y</strong></td>
<td>User-provided comments, or the null value.</td>
</tr>
</tbody>
</table>
**Table 141. SYSCAT.PACKAGES Catalog View (continued)**

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
</table>

**Note:**

1. If a function instance with dependencies is dropped, the package is put into an "inoperative" state, and it must be explicitly rebound. If any other object with dependencies is dropped, the package is put into an "invalid" state, and the system will attempt to rebind the package automatically when it is first referenced.

2. The DEFINER column is included for backwards compatibility. See OWNER.

3. The PKG_CREATE_TIME column is included for backwards compatibility. See CREATE_TIME.
Each row represents a distribution map that is used to distribute the rows of a table among the database partitions in a database partition group, based on hashing the table’s distribution.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMAP_ID</td>
<td>SMALLINT</td>
<td></td>
<td>Identifier for the distribution map.</td>
</tr>
<tr>
<td>PARTITIONMAP</td>
<td>BLOB (65536)</td>
<td></td>
<td>Distribution map, a vector of 32768 two-byte integers for a multiple partition database partition group. For a single partition database partition group, there is one entry denoting the partition number of the single partition.</td>
</tr>
</tbody>
</table>
Each row represents a user, group, or role that has been granted pass-through authorization to query a data source.

Table 143. SYSCAT.PASSTHRUAUTH Catalog View

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRANTOR</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Grantor of the privilege.</td>
</tr>
<tr>
<td>GRANTEE</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Holder of the privilege.</td>
</tr>
<tr>
<td>GRANTOR TYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>• S = Grantor is the system</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• U = Grantor is an individual user</td>
</tr>
<tr>
<td>GRANTEE TYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>• G = Grantee is a group</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• R = Grantee is a role</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• U = Grantee is an individual user</td>
</tr>
<tr>
<td>SERVERNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Name of the data source to which authorization is being granted.</td>
</tr>
</tbody>
</table>
SYSCAT.PREDICATESPECS

Each row represents a predicate specification.

Table 144. SYSCAT.PREDICATESPECS Catalog View

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FUNCSCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the function.</td>
</tr>
<tr>
<td>FUNCNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the function.</td>
</tr>
<tr>
<td>SPECIFICNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Name of the function instance.</td>
</tr>
<tr>
<td>FUNCID</td>
<td>INTEGER</td>
<td></td>
<td>Identifier for the function.</td>
</tr>
<tr>
<td>SPECID</td>
<td>SMALLINT</td>
<td></td>
<td>Number of this predicate specification.</td>
</tr>
<tr>
<td>CONTEXTOP</td>
<td>CHAR (8)</td>
<td></td>
<td>Comparison operator, one of the built-in relational operators (=, &lt;, &gt;, &gt;=, and so on).</td>
</tr>
<tr>
<td>CONTEXTEXP</td>
<td>CLOB (2M)</td>
<td></td>
<td>Constant, or an SQL expression.</td>
</tr>
<tr>
<td>FILTERTEXT</td>
<td>CLOB (32K)</td>
<td>Y</td>
<td>Text of the data filter expression.</td>
</tr>
</tbody>
</table>
Each row represents a referential integrity (foreign key) constraint.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONSTNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Name of the constraint.</td>
</tr>
<tr>
<td>TABSCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the dependent table.</td>
</tr>
<tr>
<td>TABNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the dependent table.</td>
</tr>
<tr>
<td>OWNER</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Authorization ID of the owner of the constraint.</td>
</tr>
<tr>
<td>OWNERTYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>• S = The owner is the system</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• U = The owner is an individual user</td>
</tr>
<tr>
<td>REFKEYNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Name of the parent key.</td>
</tr>
<tr>
<td>REFTABSCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the parent table.</td>
</tr>
<tr>
<td>REFTABNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the parent table.</td>
</tr>
<tr>
<td>COLCOUNT</td>
<td>SMALLINT</td>
<td></td>
<td>Number of columns in the foreign key.</td>
</tr>
<tr>
<td>DELETERULE</td>
<td>CHAR (1)</td>
<td></td>
<td>Delete rule.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• A = NO ACTION</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• C = CASCADE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = SET NULL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• R = RESTRICT</td>
</tr>
<tr>
<td>UPDATERULE</td>
<td>CHAR (1)</td>
<td></td>
<td>Update rule.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• A = NO ACTION</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• R = RESTRICT</td>
</tr>
<tr>
<td>CREATE_TIME</td>
<td>TIMESTAMP</td>
<td></td>
<td>Time at which the constraint was defined.</td>
</tr>
<tr>
<td>FK_COLNAMES</td>
<td>VARCHAR (640)</td>
<td></td>
<td>This column is no longer used and will be removed in a future release. Use</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>SYSCAT.KEYCOLUSE for this information.</td>
</tr>
<tr>
<td>PK_COLNAMES</td>
<td>VARCHAR (640)</td>
<td></td>
<td>This column is no longer used and will be removed in a future release. Use</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>SYSCAT.KEYCOLUSE for this information.</td>
</tr>
<tr>
<td>DEFINER1</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Authorization ID of the owner of the constraint.</td>
</tr>
</tbody>
</table>

Note:
1. The DEFINER column is included for backwards compatibility. See OWNER.
SYSCAT.ROLEAUTH

Each row represents a role granted to a user, group, role, or PUBLIC.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRANTOR</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Authorization ID that granted the role.</td>
</tr>
<tr>
<td>GRANTORTYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>• U = Grantor is an individual user</td>
</tr>
<tr>
<td>GRANTEE</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Authorization ID to which the role was granted.</td>
</tr>
<tr>
<td>GRANTEETYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>• G = The grantee is a group</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• R = The grantee is a role</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• U = The grantee is an individual user</td>
</tr>
<tr>
<td>ROLENAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Name of the role.</td>
</tr>
<tr>
<td>ROLEID</td>
<td>INTEGER</td>
<td></td>
<td>Identifier for the role.</td>
</tr>
<tr>
<td>ADMIN</td>
<td>CHAR (1)</td>
<td></td>
<td>Privilege to grant or revoke the role to or from others, or to comment on the role.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = Not held</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = Held</td>
</tr>
</tbody>
</table>
SYSCAT.ROLES

Each row represents a role.

Table 147. SYSCAT.ROLES Catalog View

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROLENAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Name of the role.</td>
</tr>
<tr>
<td>ROLEID</td>
<td>INTEGER</td>
<td></td>
<td>Identifier for the role.</td>
</tr>
<tr>
<td>CREATE_TIME</td>
<td>TIMESTAMP</td>
<td></td>
<td>Time when the role was created.</td>
</tr>
<tr>
<td>AUDITPOLICYID</td>
<td>INTEGER</td>
<td>Y</td>
<td>Identifier for the audit policy.</td>
</tr>
<tr>
<td>AUDITPOLICYNAME</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>Name of the audit policy.</td>
</tr>
<tr>
<td>REMARKS</td>
<td>VARCHAR (254)</td>
<td>Y</td>
<td>User-provided comments, or the null value.</td>
</tr>
</tbody>
</table>
SYSCAT.ROUTINEAUTH

Each row represents a user, group, or role that has been granted EXECUTE privilege on:
- a particular routine (function, method, or procedure) in the database that is not defined in a module
- all routines in a particular schema in the database that are not defined in a module

Table 148. SYSCAT.ROUTINEAUTH Catalog View

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRANTOR</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Grantor of the privilege. 'SYSIBM' if the privilege was granted by the system.</td>
</tr>
</tbody>
</table>
| GRANTORTYPE     | CHAR (1)           |          | • S = Grantor is the system  
|                 |                    |          | • U = Grantor is an individual user                                         |
| GRANTEE         | VARCHAR (128)      |          | Holder of the privilege.                                                    |
| GRANTEEETYPE    | CHAR (1)           |          | • G = Grantee is a group  
|                 |                    |          | • R = Grantee is a role  
|                 |                    |          | • U = Grantee is an individual user                                         |
| SCHEMA          | VARCHAR (128)      |          | Schema name of the routine.                                                 |
| SPECIFICNAME    | VARCHAR (128)      | Y        | Specific name of the routine. If SPECIFICNAME is the null value and ROUTINETYPE is not 'M', the privilege applies to all routines of the type specified in ROUTINETYPE in the schema specified in SCHEMA. If SPECIFICNAME is the null value and ROUTINETYPE is 'M', the privilege applies to all methods for the subject type specified by TYPENAME in the schema specified by TYPESCHEMA. If SPECIFICNAME is the null value, ROUTINETYPE is 'M', and both TYPENAME and TYPESCHEMA are null values, the privilege applies to all methods for all types in the schema. |
| TYPESCHEMA      | VARCHAR (128)      | Y        | Schema name of the type for the method. The null value if ROUTINETYPE is not 'M'. |
| TYPENAME        | VARCHAR (128)      | Y        | Unqualified name of the type for the method. The null value if ROUTINETYPE is not 'M'. If TYPENAME is the null value and ROUTINETYPE is 'M', the privilege applies to all methods for any subject type if they are in the schema specified by SCHEMA. |
| ROUTINETYPE     | CHAR (1)           |          | Type of the routine.  
|                 |                    |          | • F = Function  
|                 |                    |          | • M = Method  
|                 |                    |          | • P = Procedure |
| EXECUTEAUTH     | CHAR (1)           |          | Privilege to execute the routine.  
|                 |                    |          | • G = Held and grantable  
|                 |                    |          | • N = Not held  
|                 |                    |          | • Y = Held |
### Table 148. SYSCAT.ROUTINEAUTH Catalog View (continued)

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRANT_TIME</td>
<td>TIMESTAMP</td>
<td></td>
<td>Time at which the privilege was granted.</td>
</tr>
</tbody>
</table>
SYSCAT.ROUTINEDEP

Each row represents a dependency of a routine on some other object. The routine depends on the object of type BTYPE of name BNAME, so a change to the object affects the routine.

Table 149. SYSCAT.ROUTINEDEP Catalog View

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROUTINESCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the routine that has dependencies on another object.</td>
</tr>
<tr>
<td>ROUTINEMODULENAME</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>Unqualified name of the module.</td>
</tr>
<tr>
<td>SPECIFICNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Specific name of the routine that has dependencies on another object.</td>
</tr>
<tr>
<td>ROUTINEMODULEID</td>
<td>INTEGER</td>
<td>Y</td>
<td>Identifier for the module of the object that has dependencies on another object.</td>
</tr>
<tr>
<td>BTYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>Type of object on which there is a dependency. Possible values are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• A = Table alias</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• B = Trigger</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• F = Routine</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• G = Global temporary table</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• H = Hierarchy table</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• K = Package</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• L = Detached table</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = Nickname</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• O = Privilege dependency on all subtables or subviews in a table or view hierarchy</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Q = Sequence</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• R = User-defined data type</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• S = Materialized query table</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• T = Table (not typed)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• U = Typed table</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• V = View (not typed)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• W = Typed view</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• X = Index extension</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Z = XSR object</td>
</tr>
<tr>
<td>BSCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the object on which there is a dependency.</td>
</tr>
<tr>
<td>BMODULENAME</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>Unqualified name of the module to which the object on which a dependency belongs. The null value if not a module object.</td>
</tr>
<tr>
<td>BNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the object on which there is a dependency. For routines (BTYPE = 'F'), this is the specific name.</td>
</tr>
</tbody>
</table>
## SYSCAT.ROUTINEDEP

Table 149. SYSCAT.ROUTINEDEP Catalog View (continued)

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BMODULEID</td>
<td>INTEGER</td>
<td>Y</td>
<td>Identifier for the module of the object on which there is a dependency.</td>
</tr>
<tr>
<td>TABAUTH</td>
<td>SMALLINT</td>
<td>Y</td>
<td>If BTYPE = 'O', 'S', 'T', 'U', 'V', 'W', or 'v', encodes the privileges on the table or view that are required by the dependent routine; the null value otherwise.</td>
</tr>
<tr>
<td>ROUTINENAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>This column is no longer used and will be removed in a future release. See SPECIFICNAME.</td>
</tr>
</tbody>
</table>
Each row represents a routine-specific option value.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROUTINESCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the routine if ROUTINEMODULEID is null; otherwise schema name of the module to which the routine belongs.</td>
</tr>
<tr>
<td>ROUTINENAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the routine.</td>
</tr>
<tr>
<td>SPECIFICNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Name of the routine instance (might be system-generated).</td>
</tr>
<tr>
<td>OPTION</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Name of the federated routine option.</td>
</tr>
<tr>
<td>SETTING</td>
<td>VARCHAR (2048)</td>
<td></td>
<td>Value of the federated routine option.</td>
</tr>
</tbody>
</table>
SYSCAT.ROUTINEPARMOPTIONS

SYSCAT.ROUTINEPARMOPTIONS

Each row represents a routine parameter-specific option value.

Table 151. SYSCAT.ROUTINEPARMOPTIONS Catalog View

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROUTINESCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the routine if ROUTINEMODULEID is null; otherwise schema name of the module to which the routine belongs.</td>
</tr>
<tr>
<td>ROUTINENAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the routine.</td>
</tr>
<tr>
<td>SPECIFICNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Name of the routine instance (might be system-generated).</td>
</tr>
<tr>
<td>ORDINAL</td>
<td>SMALLINT</td>
<td></td>
<td>Position of the parameter within the routine signature.</td>
</tr>
<tr>
<td>OPTION</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Name of the federated routine option.</td>
</tr>
<tr>
<td>SETTING</td>
<td>VARCHAR (2048)</td>
<td></td>
<td>Value of the federated routine option.</td>
</tr>
</tbody>
</table>
SYSCAT.ROUTINEPARMS

Each row represents a parameter or the result of a routine defined in SYSCAT.ROUTINES.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROUTINESCHEMA</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>Schema name of the routine if ROUTINEMODULEID is null; otherwise schema name of the module to which the routine belongs.</td>
</tr>
<tr>
<td>ROUTINEMODULENAME</td>
<td>VARCHAR(128)</td>
<td>Y</td>
<td>Unqualified name of the module to which the routine belongs. The null value if not a module routine.</td>
</tr>
<tr>
<td>ROUTINENAME</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>Unqualified name of the routine.</td>
</tr>
<tr>
<td>ROUTINEMODULEID</td>
<td>INTEGER</td>
<td>Y</td>
<td>Identifier for the module to which the routine belongs. The null value if not a module routine.</td>
</tr>
<tr>
<td>SPECIFICNAME</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>Name of the routine instance (might be system-generated).</td>
</tr>
<tr>
<td>PARMNAME</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>Name of the parameter or result column, or the null value if no name exists.</td>
</tr>
<tr>
<td>ROWTYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>• B = Both input and output parameter</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• C = Result after casting</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• O = Output parameter</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• P = Input parameter</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• R = Result before casting</td>
</tr>
<tr>
<td>ORDINAL</td>
<td>SMALLINT</td>
<td></td>
<td>If ROWTYPE = ‘B’, ‘O’, or ‘P’, numerical position of the parameter within the routine signature, starting with 1; if ROWTYPE = ‘R’ and the routine returns a table, numerical position of a named column in the result table, starting with 1; 0 otherwise.</td>
</tr>
<tr>
<td>TYPESCHEMA</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>Schema name of the data type if TYPEMODULEID is null; otherwise schema name of the module to which the data type belongs.</td>
</tr>
<tr>
<td>TYPEMODULENAME</td>
<td>VARCHAR(128)</td>
<td>Y</td>
<td>Unqualified name of the module to which the data type of the parameter or result belongs. The null value if not a module data type.</td>
</tr>
<tr>
<td>TYPENAME</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>Unqualified name of the data type.</td>
</tr>
<tr>
<td>LOCATOR</td>
<td>CHAR (1)</td>
<td></td>
<td>• N = Parameter or result is not passed in the form of a locator</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = Parameter or result is passed in the form of a locator</td>
</tr>
<tr>
<td>LENGTH</td>
<td>INTEGER</td>
<td></td>
<td>Length of the parameter or result; 0 if the parameter or result is a user-defined data type.</td>
</tr>
<tr>
<td>Column Name</td>
<td>Data Type</td>
<td>Nullable</td>
<td>Description</td>
</tr>
<tr>
<td>----------------</td>
<td>-----------------</td>
<td>----------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>SCALE(^1)</td>
<td>SMALLINT</td>
<td></td>
<td>Scale if the parameter or result data type is DECIMAL; the number of digits of fractional seconds if the parameter or result data type is TIMESTAMP; 0 otherwise.</td>
</tr>
<tr>
<td>CODEPAGE</td>
<td>SMALLINT</td>
<td></td>
<td>Code page of this parameter or result; 0 denotes either not applicable, or a parameter or result for character data declared with the FOR BIT DATA attribute.</td>
</tr>
<tr>
<td>COLLATIONSHEMA</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>For string types, the schema name of the collation for the parameter; the null value otherwise.</td>
</tr>
<tr>
<td>COLLATIONNAME</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>For string types, the unqualified name of the collation for the parameter; the null value otherwise.</td>
</tr>
<tr>
<td>CAST_FUNCSHEMA</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>Schema name of the function used to cast an argument or a result. Applies to sourced and external functions; the null value otherwise.</td>
</tr>
<tr>
<td>CAST_FUNCSPECIFIC</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>Unqualified name of the function used to cast an argument or a result. Applies to sourced and external functions; the null value otherwise.</td>
</tr>
<tr>
<td>TARGET_TYPESCHEMA</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>Schema name of the target type if the type of the parameter or result is REFERENCE. Null value if the type of the parameter or result is not REFERENCE.</td>
</tr>
<tr>
<td>TARGET_TYPEMODULENAME</td>
<td>VARCHAR(128)</td>
<td>Y</td>
<td>Unqualified name of the module to which the target type belongs if the type of the parameter or result is REFERENCE. The null value if the type of the parameter or result is not REFERENCE or if the target type is not a module data type.</td>
</tr>
<tr>
<td>TARGET_TYPENAME</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>Unqualified name of the module to which the target type belongs if the type of the parameter or result is REFERENCE. The null value if the type of the parameter or result is not REFERENCE or if the target type is not a module data type.</td>
</tr>
<tr>
<td>SCOPE_TABSCHEMA</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>Schema name of the scope (target table) if the parameter type is REFERENCE; null value otherwise.</td>
</tr>
<tr>
<td>SCOPE_TABNAME</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>Unqualified name of the scope (target table) if the parameter type is REFERENCE; null value otherwise.</td>
</tr>
<tr>
<td>TRANSFORMGRPNAMEN</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>Name of the transform group for a structured type parameter or result.</td>
</tr>
<tr>
<td>DEFAULT</td>
<td>CLOB (64K)</td>
<td>Y</td>
<td>Expression used to calculate the default value of the parameter. The null value if DEFAULT clause was not specified for the parameter.</td>
</tr>
<tr>
<td>REMARKS</td>
<td>VARCHAR (254)</td>
<td>Y</td>
<td>User-provided comments, or the null value.</td>
</tr>
</tbody>
</table>
Table 152. SYSCAT.ROUTINEPARMS Catalog View (continued)

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
</table>

Note:
1. LENGTH and SCALE are set to 0 for sourced functions (functions defined with a reference to another function), because they inherit the length and scale of parameters from their source.
SYSCAT.ROUTINES

Each row represents a user-defined routine (scalar function, table function, sourced function, method, or procedure). Does not include built-in functions.

Table 153. SYSCAT.ROUTINES Catalog View

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROUTINESCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the routine if ROUTINEMODULEID is null; otherwise schema name of the module to which the routine belongs.</td>
</tr>
<tr>
<td>ROUTINEMODULENAME</td>
<td>VARCHAR(128)</td>
<td>Y</td>
<td>Unqualified name of the module to which the routine belongs. The null value if not a module routine.</td>
</tr>
<tr>
<td>ROUTINENAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the routine.</td>
</tr>
<tr>
<td>ROUTINETYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>Type of routine.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• F = Function</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• M = Method</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• P = Procedure</td>
</tr>
<tr>
<td>OWNER</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Authorization ID of the owner of the routine.</td>
</tr>
<tr>
<td>OWNERTYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>• S = The owner is the system</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• U = The owner is an individual user</td>
</tr>
<tr>
<td>SPECIFICNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Name of the routine instance (might be system-generated).</td>
</tr>
<tr>
<td>ROUTINEID</td>
<td>INTEGER</td>
<td></td>
<td>Identifier for the routine.</td>
</tr>
<tr>
<td>ROUTINEMODULEID</td>
<td>INTEGER</td>
<td>Y</td>
<td>Identifier for the module to which the routine belongs. The null value if not a module routine.</td>
</tr>
<tr>
<td>RETURN_TYPESCHEMA</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>Schema name of the return type for a scalar function or method.</td>
</tr>
<tr>
<td>RETURN_TYPEMODULE</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>The module name of the return type; the null value if the return type does not belong to any module.</td>
</tr>
<tr>
<td>RETURN_TYPENAME</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>Unqualified name of the return type for a scalar function or method.</td>
</tr>
<tr>
<td>ORIGIN</td>
<td>CHAR (1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• B = Built-in</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• E = User-defined, external</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• M = Template function</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• F = Federated procedure</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Q = SQL-bodied</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• R = System-generated SQL-bodied routine</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• S = System-generated</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• T = System-generated transform function (not directly invokable)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• U = User-defined, based on a source</td>
</tr>
<tr>
<td>FUNCTIONTYPE</td>
<td>CHAR (1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• C = Column or aggregate</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• R = Row</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• S = Scalar</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• T = Table</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Blank = Procedure</td>
</tr>
<tr>
<td>PARM_COUNT</td>
<td>SMALLINT</td>
<td></td>
<td>Number of routine parameters.</td>
</tr>
<tr>
<td>Column Name</td>
<td>Data Type</td>
<td>Nullable</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------</td>
<td>----------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>LANGUAGE</td>
<td>CHAR (8)</td>
<td></td>
<td>Implementation language for the routine body (or for the source function body, if this function is sourced on another function). Possible values are 'C', 'COBOL', 'JAVA', 'OLE', 'OLEDB', or 'SQL'. Blanks if ORIGIN is not 'E', 'Q', or 'R'.</td>
</tr>
</tbody>
</table>
| DIALECT      | VARCHAR(10)|          | The source dialect of the SQL routine body:  
• DB2 SQL PL  
• PL/SQL  
• Blank = Not an SQL routine |
| SOURCESCHEMA | VARCHAR (128) | Y       | If ORIGIN = 'U' and the source function is a user-defined function, contains the schema name of the specific name of the source function. If ORIGIN = 'U' and the source function is a built-in function, contains the value 'SYSIBM'. The null value if ORIGIN is not 'U'. |
| SOURCESPECIFIC | VARCHAR (128) | Y       | If ORIGIN = 'U' and the source function is a user-defined function, contains the unqualified specific name of the source function. If ORIGIN = 'U' and the source function is a built-in function, contains the value 'N/A for built-in'. The null value if ORIGIN is not 'U'. |
| PUBLISHED    | CHAR (1)    |          | Indicates whether the module routine can be invoked outside its module.  
• N = The module routine is not published  
• Y = The module routine is published  
• Blank = Not applicable |
| DETERMINISTIC | CHAR (1)    |          |  
• N = Results are not deterministic (same parameters might give different results in different routine calls)  
• Y = Results are deterministic  
• Blank = ORIGIN is not 'E', 'F', 'Q', or 'R' |
| EXTERNAL_ACTION | CHAR (1)  |          |  
• E = Function has external side-effects (therefore, the number of invocations is important)  
• N = No side-effects  
• Blank = ORIGIN is not 'E', 'F', 'Q', or 'R' |
| NULLCALL     | CHAR (1)    |          |  
• N = RETURNS NULL ON NULL INPUT (function result is implicitly the null value if one or more operands are null)  
• Y = CALLED ON NULL INPUT  
• Blank = ORIGIN is not 'E', 'Q', or 'R' |
| CAST_FUNCTION | CHAR (1)    |          |  
• N = Not a cast function  
• Y = Cast function  
• Blank = ROUTINETYPE is not 'F' |
| ASSIGN_FUNCTION | CHAR (1) |          |  
• N = Not an assignment function  
• Y = Implicit assignment function  
• Blank = ROUTINETYPE is not 'F' |
| SCRATCHPAD   | CHAR (1)    |          |  
• N = Routine has no scratchpad  
• Y = Routine has a scratchpad  
• Blank = ORIGIN is not 'E' or ROUTINETYPE is 'P' |
Table 153. SYSCAT.ROUTINES Catalog View (continued)

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCRATCHPAD_LENGTH</td>
<td>SMALLINT</td>
<td></td>
<td>Size (in bytes) of the scratchpad for the routine.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• -1 = LANGUAGE is 'OLEDB' and SCRATCHPAD is &quot;Y&quot;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• 0 = SCRATCHPAD is not 'Y'</td>
</tr>
<tr>
<td>FINALCALL</td>
<td>CHAR (1)</td>
<td></td>
<td>• N = No final call is made</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = Final call is made to this routine at the runtime end-of-statement</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Blank = ORIGIN is not 'E' or ROUTINETYPE is 'P'</td>
</tr>
<tr>
<td>PARALLEL</td>
<td>CHAR (1)</td>
<td></td>
<td>• N = Routine cannot be executed in parallel</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = Routine can be executed in parallel</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Blank = ORIGIN is not 'E'</td>
</tr>
<tr>
<td>PARAMETER_STYLE</td>
<td>CHAR (8)</td>
<td></td>
<td>Parameter style that was declared when the routine was created. Possible values are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• DB2DARI</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• DB2GENRL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• DB2SQL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• GENERAL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• GNRLNULL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• JAVA</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• SQL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Blanks if ORIGIN is not 'E'</td>
</tr>
<tr>
<td>FENCED</td>
<td>CHAR (1)</td>
<td></td>
<td>• N = Not fenced</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = Fenced</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Blank = ORIGIN is not 'E'</td>
</tr>
<tr>
<td>SQL_DATA_ACCESS</td>
<td>CHAR (1)</td>
<td></td>
<td>Indicates what type of SQL statements, if any, the database manager should assume is contained in the routine.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• C = Contains SQL (simple expressions with no subqueries only)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• M = Contains SQL statements that modify data</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = Does not contain SQL statements</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• R = Contains read-only SQL statements</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Blank = ORIGIN is not 'E', 'F', 'Q', or 'R'</td>
</tr>
<tr>
<td>DBINFO</td>
<td>CHAR (1)</td>
<td></td>
<td>Indicates whether a DBINFO parameter is passed to an external routine.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = DBINFO is not passed</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = DBINFO is passed</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Blank = ORIGIN is not 'E'</td>
</tr>
<tr>
<td>PROGRAMTYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>Indicates how the external routine is invoked.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• M = Main</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• S = Subroutine</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Blank = ORIGIN is 'F'</td>
</tr>
<tr>
<td>COMMIT_ON_RETURN</td>
<td>CHAR (1)</td>
<td></td>
<td>Indicates whether the transaction is committed on successful return from this procedure.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = The unit of work is not committed</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = The unit of work is committed</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Blank = ROUTINETYPE is not 'P'</td>
</tr>
</tbody>
</table>
### Table 153. SYSCAT.ROUTINES Catalog View (continued)

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AUTONOMOUS</td>
<td>CHAR (1)</td>
<td></td>
<td>Indicate whether or not the routine executes autonomously.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- ( N ) = Routine does not execute autonomously from invoking transaction</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- ( Y ) = Routine executes autonomously from invoking transaction</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Blank = ROUTINETYPE is not ‘P’</td>
</tr>
<tr>
<td>RESULT_SETS</td>
<td>SMALLINT</td>
<td></td>
<td>Estimated maximum number of result sets.</td>
</tr>
<tr>
<td>SPEC_REG</td>
<td>CHAR (1)</td>
<td></td>
<td>Indicate the special registers values that are used when the routine is called.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- ( I ) = Inherited special registers</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Blank = PARAMETER_STYLE is ‘DB2DARI’ or ORIGIN is not ‘E’, ‘Q’, or ‘R’</td>
</tr>
<tr>
<td>FEDERATED</td>
<td>CHAR (1)</td>
<td></td>
<td>Indicate whether or not federated objects can be accessed from the routine.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- ( Y ) = Federated objects can be accessed</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Blank = ORIGIN is not ‘F’</td>
</tr>
<tr>
<td>THREADSAFE</td>
<td>CHAR (1)</td>
<td></td>
<td>Indicate whether or not the routine can run in the same process as other routines.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- ( N ) = Routine is not threadsafe</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- ( Y ) = Routine is threadsafe</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Blank = ORIGIN is not ‘E’</td>
</tr>
<tr>
<td>VALID</td>
<td>CHAR (1)</td>
<td></td>
<td>Applies to LANGUAGE = ‘SQL’ and routines having parameters with default; blank otherwise.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- ( N ) = Routine needs rebinding</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- ( X ) = Routine is inoperative and must be recreated</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- ( Y ) = Routine is valid</td>
</tr>
<tr>
<td>MODULEROUTINEIMPLEMENTED</td>
<td>CHAR (1)</td>
<td></td>
<td>Indicate whether the module routine body is implemented.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- ( N ) = Module routine body is not implemented</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- ( Y ) = Module routine body is implemented</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Blank = ROUTINEMODULENAME is null value</td>
</tr>
<tr>
<td>METHODIMPLEMENTED</td>
<td>CHAR (1)</td>
<td></td>
<td>Indicate whether the method body is implemented.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- ( N ) = Method body is not implemented</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- ( Y ) = Method body is implemented</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Blank = ROUTINETYPE is not ‘M’ or ROUTINEMODULENAME is not the null value</td>
</tr>
<tr>
<td>METHODEFFECT</td>
<td>CHAR (2)</td>
<td></td>
<td>Indicate the method effect.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- ( CN ) = Constructor method</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- ( MU ) = Mutator method</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- ( OB ) = Observer method</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Blanks = Not a system method</td>
</tr>
<tr>
<td>TYPE_PRESERVING</td>
<td>CHAR (1)</td>
<td></td>
<td>Indicate whether the return type is type-preserving.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- ( N ) = Return type is the declared return type of the method</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- ( Y ) = Return type is governed by a &quot;type-preserving&quot; parameter; all system-generated mutator methods are type-preserving</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Blank = ROUTINETYPE is not ‘M’</td>
</tr>
</tbody>
</table>
### Table 153. SYSCAT.ROUTINES Catalog View (continued)

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WITH_FUNC_ACCESS</td>
<td>CHAR (1)</td>
<td></td>
<td>• N = This method cannot be invoked by using functional notation</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = This method can be invoked by using functional notation; that is, the &quot;WITH FUNCTION ACCESS&quot; attribute is specified</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Blank = ROUTINETYPE is not 'M'</td>
</tr>
<tr>
<td>OVERRIDDEN_METHODID</td>
<td>INTEGER</td>
<td>Y</td>
<td>Identifier for the overridden method when the OVERRIDING option is specified for a user-defined method. The null value if ROUTINETYPE is not 'M'.</td>
</tr>
<tr>
<td>SUBJECT_TYPESCHEMA</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>Schema name of the subject type for the user-defined method. The null value if ROUTINETYPE is not 'M'.</td>
</tr>
<tr>
<td>SUBJECT_TYPENAME</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>Unqualified name of the subject type for the user-defined method. The null value if ROUTINETYPE is not 'M'.</td>
</tr>
<tr>
<td>CLASS</td>
<td>VARCHAR (384)</td>
<td>Y</td>
<td>For LANGUAGE JAVA, CLR, or OLE, this is the class that implements this routine; null value otherwise.</td>
</tr>
<tr>
<td>JAR_ID</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>For LANGUAGE JAVA, this is the JAR_ID of the installed jar file that implements this routine if a jar file was specified at routine creation time; null value otherwise. For LANGUAGE CLR, this is the assembly file that implements this routine.</td>
</tr>
<tr>
<td>JARSCHEMA</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>For LANGUAGE JAVA when a JAR_ID is present, this is the schema name of the jar file that implements this routine; null value otherwise.</td>
</tr>
<tr>
<td>JAR_SIGNATURE</td>
<td>VARCHAR (2048)</td>
<td>Y</td>
<td>For LANGUAGE JAVA, this is the method signature of this routine's specified Java method. For LANGUAGE CLR, this is a reference field for this CLR routine. Null value otherwise.</td>
</tr>
<tr>
<td>CREATE_TIME</td>
<td>TIMESTAMP</td>
<td></td>
<td>Time at which the routine was created.</td>
</tr>
<tr>
<td>ALTER_TIME</td>
<td>TIMESTAMP</td>
<td></td>
<td>Time at which the routine was last altered.</td>
</tr>
<tr>
<td>FUNC_PATH</td>
<td>CLOB (2K)</td>
<td>Y</td>
<td>SQL path in effect when the routine was defined. The null value if LANGUAGE is not 'SQL'.</td>
</tr>
<tr>
<td>QUALIFIER</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Value of the default schema at the time of object definition. Used to complete any unqualified references.</td>
</tr>
<tr>
<td>IOS_PER_INVOC</td>
<td>DOUBLE</td>
<td></td>
<td>Estimated number of inputs/outputs (I/Os) per invocation; 0 is the default; -1 if not known.</td>
</tr>
<tr>
<td>INSTS_PER_INVOC</td>
<td>DOUBLE</td>
<td></td>
<td>Estimated number of instructions per invocation; 450 is the default; -1 if not known.</td>
</tr>
<tr>
<td>IOS_PER_ARGBYTE</td>
<td>DOUBLE</td>
<td></td>
<td>Estimated number of I/Os per input argument byte; 0 is the default; -1 if not known.</td>
</tr>
<tr>
<td>INSTS_PER_ARGBYTE</td>
<td>DOUBLE</td>
<td></td>
<td>Estimated number of instructions per input argument byte; 0 is the default; -1 if not known.</td>
</tr>
<tr>
<td>PERCENT_ARGBYTES</td>
<td>SMALLINT</td>
<td></td>
<td>Estimated average percent of input argument bytes that the routine will actually read; 100 is the default; -1 if not known.</td>
</tr>
<tr>
<td>INITIAL_IOS</td>
<td>DOUBLE</td>
<td></td>
<td>Estimated number of I/Os performed the first time that the routine is invoked; 0 is the default; -1 if not known.</td>
</tr>
<tr>
<td>INITIAL_INSTS</td>
<td>DOUBLE</td>
<td></td>
<td>Estimated number of instructions executed the first time the routine is invoked; 0 is the default; -1 if not known.</td>
</tr>
</tbody>
</table>
## SYSCAT.ROUTINES

### Table 153. SYSCAT.ROUTINES Catalog View (continued)

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CARDINALITY</td>
<td>BIGINT</td>
<td></td>
<td>Predicted cardinality of a table function; -1 if not known, or if the routine is not a table function.</td>
</tr>
<tr>
<td>SELECTIVITY²</td>
<td>DOUBLE</td>
<td></td>
<td>For user-defined predicates; -1 if there are no user-defined predicates.</td>
</tr>
<tr>
<td>RESULT_COLS</td>
<td>SMALLINT</td>
<td></td>
<td>For a table function (ROUTINETYPE = 'F' and FUNCTIONTYPE = 'T'), contains the number of columns in the result table; for a procedure (ROUTINETYPE = 'P'), contains 0; contains 1 otherwise.</td>
</tr>
<tr>
<td>IMPLEMENTATION</td>
<td>VARCHAR (762)</td>
<td>Y</td>
<td>If ORIGIN = 'E', identifies the path/module/function that implements this function. If ORIGIN = 'U' and the source function is built-in, this column contains the name and signature of the source function. Null value otherwise.</td>
</tr>
<tr>
<td>LIB_ID</td>
<td>INTEGER</td>
<td>Y</td>
<td>Reserved for future use.</td>
</tr>
<tr>
<td>TEXT BODY OFFSET</td>
<td>INTEGER</td>
<td></td>
<td>If LANGUAGE = 'SQL', the offset to the start of the compiled SQL routine body in the full text of the CREATE statement; -1 if LANGUAGE is not 'SQL' or the SQL routine is inlined.</td>
</tr>
<tr>
<td>TEXT</td>
<td>CLOB (2M)</td>
<td>Y</td>
<td>If LANGUAGE = 'SQL', the full text of the CREATE FUNCTION, CREATE METHOD, or CREATE PROCEDURE statement; null value otherwise.</td>
</tr>
</tbody>
</table>
| NEWSAVEPOINTLEVEL   | CHAR (1)    |          | Indicates whether the routine initiates a new savepoint level when it is invoked.  
|                     |             |          | • N = A new savepoint level is not initiated when the routine is invoked; the routine uses the existing savepoint level  
|                     |             |          | • Y = A new savepoint level is initiated when the routine is invoked  
|                     |             |          | • Blank = Not applicable |
| DEBUG_MODE³         | VARCHAR (8) |          | Indicates whether or not the routine can be debugged using the DB2 debugger.  
|                     |             |          | • DISALLOW = Routine is not debuggable  
|                     |             |          | • ALLOW = Routine is debuggable, and can participate in a client debug session with the DB2 debugger  
|                     |             |          | • DISABLE = Routine is not debuggable, and this setting cannot be altered without dropping and recreating the routine  
|                     |             |          | • Blank = Routine type is not currently supported by the DB2 debugger |
| TRACE_LEVEL         | VARCHAR (1) | Y        | Reserved for future use.                                                    |
| DIAGNOSTIC_LEVEL    | VARCHAR (1) | Y        | Reserved for future use.                                                    |
| CHECKOUT_USERID     | VARCHAR (128) | Y       | ID of the user who performed a checkout of the object; the null value if the object is not checked out. |
| PRECOMPILE_OPTIONS  | VARCHAR (1024) | Y       | The precompile and bind options that were in effect when the compiled SQL routine was created. The null value if LANGUAGE is not 'SQL' or if the SQL routine is not compiled. |
### Table 153. SYSCAT.ROUTINES Catalog View (continued)

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMPILE_OPTIONS</td>
<td>VARCHAR (1024)</td>
<td>Y</td>
<td>The value of the SQL_CCFLAGS special register that was in effect when the compiled SQL routine was created and inquiry directives were present. An empty string if no inquiry directives were present in the compiled SQL routine. The null value if LANGUAGE is not 'SQL' or if the SQL routine is not compiled.</td>
</tr>
<tr>
<td>EXECUTION_CONTROL</td>
<td>CHAR (1)</td>
<td></td>
<td>Execution control mode of a common language runtime (CLR) routine. Possible values are: \n</td>
</tr>
<tr>
<td>CODEPAGE</td>
<td>SMALLINT</td>
<td></td>
<td>Routine code page, which specifies the default code page used for all character parameter types, result types, and local variables within the routine body.</td>
</tr>
<tr>
<td>COLLATIONSCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the collation for the routine.</td>
</tr>
<tr>
<td>COLLATIONNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the collation for the routine.</td>
</tr>
<tr>
<td>COLLATIONSCHEMA_ORDERBY</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the collation for ORDER BY clauses in the routine.</td>
</tr>
<tr>
<td>COLLATIONNAME_ORDERBY</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the collation for ORDER BY clauses in the routine.</td>
</tr>
<tr>
<td>ENCODING_SCHEME</td>
<td>CHAR (1)</td>
<td></td>
<td>Encoding scheme of the routine, as specified in the PARAMETER CCSID clause. Possible values are: \n</td>
</tr>
<tr>
<td>LAST_REGEN_TIME</td>
<td>TIMESTAMP</td>
<td></td>
<td>Time at which the SQL routine packed descriptor was last regenerated.</td>
</tr>
<tr>
<td>INHERITLOCKREQUEST</td>
<td>CHAR (1)</td>
<td></td>
<td>• N = This function or method cannot be invoked in the context of an SQL statement that includes a lock-request-clause as part of a specified isolation-clause \n</td>
</tr>
<tr>
<td>DEFINER</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Authorization ID of the owner of the routine.</td>
</tr>
<tr>
<td>REMARKS</td>
<td>VARCHAR (254)</td>
<td>Y</td>
<td>User-provided comments, or the null value.</td>
</tr>
</tbody>
</table>

**Note:**

1. For SQL procedures created before Version 8.2 and upgraded to Version 9, 'E' (instead of 'Q').
2. During database upgrade, the SELECTIVITY column will be set to -1 in the packed descriptor and system catalogs for all user-defined routines. For a user-defined predicate, the selectivity in the system catalog will be -1. In this case, the selectivity value used by the optimizer is 0.01.
3. For Java routines, the DEBUG_MODE setting does not indicate whether the Java routine was actually compiled in debug mode, or whether a debug Jar was installed at the server.
4. The DEFINER column is included for backwards compatibility. See OWNER.
SYSCAT.ROUTINESFEDERATED

Each row represents a federated procedure.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROUTINESCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the routine if ROUTINEMODULEID is null; otherwise schema name of the module to which the routine belongs.</td>
</tr>
<tr>
<td>ROUTINENAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the routine.</td>
</tr>
<tr>
<td>ROUTINETYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>Type of routine.</td>
</tr>
<tr>
<td>OWNER</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Authorization ID of the owner of the routine.</td>
</tr>
<tr>
<td>OWNERTYPE</td>
<td>CHAR (1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPECIFICNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Name of the routine instance (might be system-generated).</td>
</tr>
<tr>
<td>ROUTINEID</td>
<td>INTEGER</td>
<td></td>
<td>Identifier for the routine.</td>
</tr>
<tr>
<td>PARM_COUNT</td>
<td>SMALLINT</td>
<td></td>
<td>Number of routine parameters.</td>
</tr>
<tr>
<td>DETERMINISTIC</td>
<td>CHAR (1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXTERNAL_ACTION</td>
<td>CHAR (1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQL_DATA_ACCESS</td>
<td>CHAR (1)</td>
<td></td>
<td>Indicates what type of SQL statements, if any, the database manager should assume is contained in the routine.</td>
</tr>
<tr>
<td>COMMIT_ON_RETURN</td>
<td>CHAR (1)</td>
<td></td>
<td>Indicates whether the transaction is committed on successful return from this procedure.</td>
</tr>
<tr>
<td>RESULT_SETS</td>
<td>SMALLINT</td>
<td></td>
<td>Estimated maximum number of result sets.</td>
</tr>
<tr>
<td>CREATE_TIME</td>
<td>TIMESTAMP</td>
<td></td>
<td>Time at which the routine was created.</td>
</tr>
<tr>
<td>ALTER_TIME</td>
<td>TIMESTAMP</td>
<td></td>
<td>Time at which the routine was last altered.</td>
</tr>
</tbody>
</table>
### SYSCAT.ROUTINESFEDERATED

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>QUALIFIER</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Value of the default schema at the time of object definition. Used to complete any unqualified references.</td>
</tr>
<tr>
<td>RESULT_COLS</td>
<td>SMALLINT</td>
<td></td>
<td>For a procedure (ROUTINETYPE = 'P'), contains 0; contains 1 otherwise.</td>
</tr>
<tr>
<td>CODEPAGE</td>
<td>SMALLINT</td>
<td></td>
<td>Routine code page, which specifies the default code page used for all character parameter types, result types, and local variables within the routine body.</td>
</tr>
<tr>
<td>LAST_REGEN_TIME</td>
<td>TIMESTAMP</td>
<td></td>
<td>Time at which the SQL routine packed descriptor was last regenerated.</td>
</tr>
<tr>
<td>REMOTE_PROCEDURE</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>Unqualified name of the source procedure for which the federated routine was created.</td>
</tr>
<tr>
<td>REMOTE_SCHEMA</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>Schema name of the source procedure for which the federated routine was created.</td>
</tr>
<tr>
<td>SERVERNAME</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>Name of the data source that contains the source procedure for which the federated routine was created.</td>
</tr>
<tr>
<td>REMOTE_PACKAGE</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>Name of the package to which the source procedure belongs (applies only to wrappers for Oracle data sources).</td>
</tr>
<tr>
<td>REMOTE_PROCEDURE_</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>Reserved for future use.</td>
</tr>
<tr>
<td>ALTER_TIME</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>REMARKS</td>
<td>VARCHAR (254)</td>
<td>Y</td>
<td>User-provided comments, or the null value.</td>
</tr>
</tbody>
</table>
SYSCAT.ROWFIELDS

Each row represents a field that is defined for a user-defined row data type.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TYPESCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the row data type that includes the field.</td>
</tr>
<tr>
<td>TYPEMODULENAME</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>Unqualified name of the module to which the row data type belongs. The null value if not a module row data type.</td>
</tr>
<tr>
<td>TYPENAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the row data type that includes the field.</td>
</tr>
<tr>
<td>FIELDNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Field name.</td>
</tr>
<tr>
<td>FIELDTYPESCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the data type of the field.</td>
</tr>
<tr>
<td>FIELDTYPENAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the data type of the field.</td>
</tr>
<tr>
<td>ORDINAL</td>
<td>SMALLINT</td>
<td></td>
<td>Position of the field in the definition of the row data type, starting with 0.</td>
</tr>
<tr>
<td>LENGTH</td>
<td>INTEGER</td>
<td></td>
<td>Length of the field data type. For decimal types, contains the precision.</td>
</tr>
<tr>
<td>SCALE</td>
<td>SMALLINT</td>
<td></td>
<td>For decimal types, contains the scale of the field data type; for timestamp types, contains the timestamp precision of the field data type; 0 otherwise.</td>
</tr>
<tr>
<td>CODEPAGE</td>
<td>SMALLINT</td>
<td></td>
<td>For string types, denotes the code page; 0 indicates FOR BIT DATA; 0 for non-string types.</td>
</tr>
<tr>
<td>COLLATIONSCHEMA</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>For string types, the schema name of the collation for the field; the null value otherwise.</td>
</tr>
<tr>
<td>COLLATIONNAME</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>For string types, the unqualified name of the collation for the field; the null value otherwise.</td>
</tr>
</tbody>
</table>
Each row represents a user, group, or role that has been granted one or more privileges on a schema.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRANTOR</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Grantor of a privilege.</td>
</tr>
<tr>
<td>GRANTORTYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>• S = Grantor is the system</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• U = Grantor is an individual user</td>
</tr>
<tr>
<td>GRANTEE</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Holder of a privilege.</td>
</tr>
<tr>
<td>GRANTEEETYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>• G = Grantee is a group</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• R = Grantee is a role</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• U = Grantee is an individual user</td>
</tr>
<tr>
<td>SCHEMANAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Name of the schema to which this privilege applies.</td>
</tr>
<tr>
<td>ALTERINAUTH</td>
<td>CHAR (1)</td>
<td></td>
<td>Privilege to alter or comment on objects in the named schema.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• G = Held and grantable</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = Not held</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = Held</td>
</tr>
<tr>
<td>CREATEINAUTH</td>
<td>CHAR (1)</td>
<td></td>
<td>Privilege to create objects in the named schema.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• G = Held and grantable</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = Not held</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = Held</td>
</tr>
<tr>
<td>DROPINAUTH</td>
<td>CHAR (1)</td>
<td></td>
<td>Privilege to drop objects from the named schema.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• G = Held and grantable</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = Not held</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = Held</td>
</tr>
</tbody>
</table>
Each row represents a schema.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCHEMANAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Name of the schema.</td>
</tr>
<tr>
<td>OWNER</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Authorization ID of the owner of the schema.</td>
</tr>
<tr>
<td>OWNERTYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>• S = The owner is the system</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• U = The owner is an individual user</td>
</tr>
<tr>
<td>DEFINER</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Authorization ID of the definer of the schema or authorization ID of the</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>owner of the schema if the ownership of the schema has been transferred.</td>
</tr>
<tr>
<td>DEFINERTYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>• S = The definer is the system</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• U = The definer is an individual user</td>
</tr>
<tr>
<td>CREATE_TIME</td>
<td>TIMESTAMP</td>
<td></td>
<td>Time at which the schema was created.</td>
</tr>
<tr>
<td>REMARKS</td>
<td>VARCHAR (254)</td>
<td>Y</td>
<td>User-provided comments, or the null value.</td>
</tr>
</tbody>
</table>
SYSCAT.SECURITYLABELACCESS

Each row represents a security label that was granted to the database authorization ID.

*Table 158. SYSCAT.SECURITYLABELACCESS Catalog View*

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRANTOR</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Grantor of the security label.</td>
</tr>
<tr>
<td>GRANTEE</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Holder of the security label.</td>
</tr>
</tbody>
</table>
| GRANTEEETYPE  | CHAR (1)      |          | • G = Grantee is a group
• R = Grantee is a role
• U = Grantee is an individual user |
| SECLABELID    | INTEGER       |          | Identifier for the security label. For the name of the security label, select the SECLABELNAME column for the corresponding SECLABELID value in the SYSCAT.SECURITYLABELS catalog view. |
| SECPOLICYID   | INTEGER       |          | Identifier for the security policy that is associated with the security label. For the name of the security policy, select the SECPOLICYNAME column for the corresponding SECPOLICYID value in the SYSCAT.SECURITYPOLICIES catalog view. |
| ACCESSTYPE    | CHAR (1)      |          | • B = Both read and write access
• R = Read access
• W = Write access |
| GRANT_TIME    | TIMESTAMP     |          | Time at which the security label was granted.                              |
Each row represents an element value for a security label component.

Table 159. SYSCAT.SECURITYLABELCOMPONENTELEMENTS Catalog View

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMPID</td>
<td>INTEGER</td>
<td></td>
<td>Identifier for the security label component.</td>
</tr>
<tr>
<td>ELEMENTVALUE</td>
<td>VARCHAR (32)</td>
<td></td>
<td>Element value for the security label component.</td>
</tr>
<tr>
<td>ELEMENTVALUEENCODING</td>
<td>CHAR (8) FOR BIT DATA</td>
<td></td>
<td>Encoded form of the element value.</td>
</tr>
<tr>
<td>PARENTELEMENTVALUE</td>
<td>VARCHAR (32)</td>
<td>Y</td>
<td>Name of the parent of an element for tree components; the null value for set and array components, and for the ROOT node of a tree component.</td>
</tr>
</tbody>
</table>
SYSCAT.SECURITYLABELCOMPONENTS

Each row represents a security label component.

Table 160. SYSCAT.SECURITYLABELCOMPONENTS Catalog View

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMPNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Name of the security label component.</td>
</tr>
<tr>
<td>COMPID</td>
<td>INTEGER</td>
<td></td>
<td>Identifier for the security label component.</td>
</tr>
<tr>
<td>COMPTYYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>Security label component type.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• A = Array</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• S = Set</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• T = Tree</td>
</tr>
<tr>
<td>NUMELEMENTS</td>
<td>INTEGER</td>
<td></td>
<td>Number of elements in the security label component.</td>
</tr>
<tr>
<td>CREATE_TIME</td>
<td>TIMESTAMP</td>
<td></td>
<td>Time at which the security label component was created.</td>
</tr>
<tr>
<td>REMARKS</td>
<td>VARCHAR (254)</td>
<td></td>
<td>User-provided comments, or the null value.</td>
</tr>
</tbody>
</table>
Each row represents a security label.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SECLABELNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Name of the security label.</td>
</tr>
<tr>
<td>SECLABELID</td>
<td>INTEGER</td>
<td></td>
<td>Identifier for the security label.</td>
</tr>
<tr>
<td>SECPOICYID</td>
<td>INTEGER</td>
<td></td>
<td>Identifier for the security policy to which the security label belongs.</td>
</tr>
<tr>
<td>SECLABEL</td>
<td>SYSPROC.DB2SECURITYLABEL</td>
<td></td>
<td>Internal representation of the security label.</td>
</tr>
<tr>
<td>CREATE_TIME</td>
<td>TIMESTAMP</td>
<td></td>
<td>Time at which the security label was created.</td>
</tr>
<tr>
<td>REMARKS</td>
<td>VARCHAR (254)</td>
<td>Y</td>
<td>User-provided comments, or the null value.</td>
</tr>
</tbody>
</table>
SYSCAT.SECURITYPOLICIES

Each row represents a security policy.

Table 162. SYSCAT.SECURITYPOLICIES Catalog View

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SECPOLICYNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Name of the security policy.</td>
</tr>
<tr>
<td>SECPOLICYID</td>
<td>INTEGER</td>
<td></td>
<td>Identifier for the security policy.</td>
</tr>
<tr>
<td>NUMSECLABELCOMP</td>
<td>INTEGER</td>
<td></td>
<td>Number of security label components in the security policy.</td>
</tr>
<tr>
<td>RWSECLABELREL</td>
<td>CHAR (1)</td>
<td></td>
<td>Relationship between the security labels for read and write access granted to the same authorization ID.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• S = The security label for write access granted to a user is a subset of the security label for read access granted to that same user</td>
</tr>
<tr>
<td>NOTAUTHWRITESECLABEL</td>
<td>CHAR (1)</td>
<td></td>
<td>Action to take when a user is not authorized to write the security label that is specified in the INSERT or UPDATE statement.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• O = Override</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• R = Restrict</td>
</tr>
<tr>
<td>CREATE_TIME</td>
<td>TIMESTAMP</td>
<td></td>
<td>Time at which the security policy was created.</td>
</tr>
<tr>
<td>GROUPAUTHS</td>
<td>CHAR (1)</td>
<td></td>
<td>Indicates if authorizations of security labels and exemptions granted to an authorization ID that represents a group will be used or ignored.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• I = Ignored</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• U = Used</td>
</tr>
<tr>
<td>ROLEAUTHS</td>
<td>CHAR (1)</td>
<td></td>
<td>Indicates if authorizations of security labels and exemptions granted to an authorization ID that represents a role will be used or ignored.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• I = Ignored</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• U = Used</td>
</tr>
<tr>
<td>REMARKS</td>
<td>VARCHAR (254)</td>
<td>Y</td>
<td>User-provided comments, or the null value.</td>
</tr>
</tbody>
</table>
SYSCAT.SECURITYPOLICYCOMPONENTRULES

Each row represents the read and write access rules for a security label component of the security policy.

Table 163. SYSCAT.SECURITYPOLICYCOMPONENTRULES Catalog View

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SECPOLICYID</td>
<td>INTEGER</td>
<td></td>
<td>Identifier for the security policy.</td>
</tr>
<tr>
<td>COMPID</td>
<td>INTEGER</td>
<td></td>
<td>Identifier for the security label component of the security policy.</td>
</tr>
<tr>
<td>ORDINAL</td>
<td>INTEGER</td>
<td></td>
<td>Position of the security label component as it appears in the security policy, starting with 1.</td>
</tr>
<tr>
<td>READACCESSRULENAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Name of the read access rule that is associated with the security label component.</td>
</tr>
<tr>
<td>READACCESSRULETEXT</td>
<td>VARCHAR (512)</td>
<td></td>
<td>Text of the read access rule that is associated with the security label component.</td>
</tr>
<tr>
<td>WRITEACCESSRULENAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Name of the write access rule that is associated with the security label component.</td>
</tr>
<tr>
<td>WRITEACCESSRULETEXT</td>
<td>VARCHAR (512)</td>
<td></td>
<td>Text of the write access rule that is associated with the security label component.</td>
</tr>
</tbody>
</table>
Each row represents a security policy exemption that was granted to a database authorization ID.

Table 164. SYSCAT.SECURITYPOLICYEXEMPTIONS Catalog View

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRANTOR</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Grantor of the exemption.</td>
</tr>
<tr>
<td>GRANTEE</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Holder of the exemption.</td>
</tr>
</tbody>
</table>
| GRANTEE TYPE    | CHAR (1)           |          | • G = Grantee is a group  
• R = Grantee is a role  
• U = Grantee is an individual user |
| SECPOLICYID     | INTEGER            |          | Identifier for the security policy for which the exemption was granted. For the name of the security policy, select the SECPOLICYNAME column for the corresponding SECPOLICYID value in the SYSCAT.SECURITYPOLICIES catalog view. |
| ACCESSRULENAME  | VARCHAR (128)      |          | Name of the access rule for which the exemption was granted.                |
| ACCESTYPE       | CHAR (1)           |          | Type of access to which the rule applies.  
• R = Read access  
• W = Write access |
| ORDINAL         | INTEGER            |          | Position of the security label component in the security policy to which the rule applies. |
| ACTIONALLOWED   | CHAR (1)           |          | If the rule is DB2LBACWRITEARRAY, then:  
• D = Write down  
• U = Write up  
Blank otherwise. |
| GRANT_TIME      | TIMESTAMP          |          | Time at which the exemption was granted.                                   |
Each row represents a user, group, or role that has been granted one or more privileges on a sequence.

Table 165. SYSCAT.SEQUENCEAUTH Catalog View

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRANTOR</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Grantor of a privilege.</td>
</tr>
<tr>
<td>GRANTORTYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>• S = Grantor is the system</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• U = Grantor is an individual user</td>
</tr>
<tr>
<td>GRANTEE</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Holder of a privilege.</td>
</tr>
<tr>
<td>GRANTEEETYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>• G = Grantee is a group</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• R = Grantee is a role</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• U = Grantee is an individual user</td>
</tr>
<tr>
<td>SEQSCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the sequence.</td>
</tr>
<tr>
<td>SEQNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the sequence.</td>
</tr>
<tr>
<td>ALTERAUTH</td>
<td>CHAR (1)</td>
<td></td>
<td>Privilege to alter the sequence.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• G = Held and grantable</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = Not held</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = Held</td>
</tr>
<tr>
<td>USAGEAUTH</td>
<td>CHAR (1)</td>
<td></td>
<td>Privilege to reference the sequence.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• G = Held and grantable</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = Not held</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = Held</td>
</tr>
</tbody>
</table>
SYSCAT.SEQUENCES

Each row represents a sequence or alias.

Table 166. SYSCAT.SEQUENCES Catalog View

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEQSCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the sequence.</td>
</tr>
<tr>
<td>SEQNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the sequence.</td>
</tr>
<tr>
<td>DEFINER¹</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Authorization ID of the owner of the sequence.</td>
</tr>
<tr>
<td>DEFINERTYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>• S = The definer is the system</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• U = The definer is an individual user</td>
</tr>
<tr>
<td>OWNER</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Authorization ID of the owner of the sequence.</td>
</tr>
<tr>
<td>OWNERTYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>• S = The owner is the system</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• U = The owner is an individual user</td>
</tr>
<tr>
<td>SEQID</td>
<td>INTEGER</td>
<td></td>
<td>Identifier for the sequence or alias.</td>
</tr>
<tr>
<td>SEQTYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>Type of sequence.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• A = Alias</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• I = Identity sequence</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• S = Sequence</td>
</tr>
<tr>
<td>BASE_SEQSCHEMA</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>If SEQTYPE is ‘A’, contains the schema name of the sequence or alias that</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>is referenced by this alias; the null value otherwise.</td>
</tr>
<tr>
<td>BASE_SEQNAME</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>If SEQTYPE is ‘A’, contains the unqualified name of the sequence or alias</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>that is referenced by this alias; the null value otherwise.</td>
</tr>
<tr>
<td>INCREMENT</td>
<td>DECIMAL (31,0)</td>
<td>Y</td>
<td>Increment value. The null value if the sequence is an alias.</td>
</tr>
<tr>
<td>START</td>
<td>DECIMAL (31,0)</td>
<td>Y</td>
<td>Start value of the sequence. The null value if the sequence is an alias.</td>
</tr>
<tr>
<td>MAXVALUE</td>
<td>DECIMAL (31,0)</td>
<td>Y</td>
<td>Maximum value of the sequence. The null value if the sequence is an alias.</td>
</tr>
<tr>
<td>MINVALUE</td>
<td>DECIMAL (31,0)</td>
<td>Y</td>
<td>Minimum value of the sequence. The null value if the sequence is an alias.</td>
</tr>
<tr>
<td>NEXTCACHEFIRSTVALUE</td>
<td>DECIMAL (31,0)</td>
<td>Y</td>
<td>The first value available to be assigned in the next cache block. If no</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>caching, the next value available to be assigned.</td>
</tr>
<tr>
<td>CYCLE</td>
<td>CHAR (1)</td>
<td></td>
<td>Indicates whether or not the sequence can continue to generate values after</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>reaching its maximum or minimum value.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = Sequence cannot cycle</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = Sequence can cycle</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Blank = Sequence is an alias</td>
</tr>
</tbody>
</table>
Table 166. SYSCAT.SEQUENCES Catalog View (continued)

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CACHE</td>
<td>INTEGER</td>
<td></td>
<td>Number of sequence values to pre-allocate in memory for faster access. 0 indicates that values of the sequence are not to be preallocated. In a partitioned database, this value applies to each database partition. -1 if the sequence is an alias.</td>
</tr>
<tr>
<td>ORDER</td>
<td>CHAR (1)</td>
<td></td>
<td>Indicates whether or not the sequence numbers must be generated in order of request.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = Sequence numbers are not required to be generated in order of request</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = Sequence numbers must be generated in order of request</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Blank = Sequence is an alias.</td>
</tr>
<tr>
<td>DATATYPEID</td>
<td>INTEGER</td>
<td></td>
<td>For built-in types, the internal identifier of the built-in type. For distinct types, the internal identifier of the distinct type. 0 if the sequence is an alias.</td>
</tr>
<tr>
<td>SOURCETYPEID</td>
<td>INTEGER</td>
<td></td>
<td>For a built-in type or if the sequence is an alias, this has a value of 0. For a distinct type, this is the internal identifier of the built-in type that is the source type for the distinct type.</td>
</tr>
<tr>
<td>CREATE_TIME</td>
<td>TIMESTAMP</td>
<td></td>
<td>Time at which the sequence was created.</td>
</tr>
<tr>
<td>ALTER_TIME</td>
<td>TIMESTAMP</td>
<td></td>
<td>Time at which the sequence was last altered.</td>
</tr>
<tr>
<td>PRECISION</td>
<td>SMALLINT</td>
<td></td>
<td>Precision of the data type of the sequence. Possible values are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• 5 = SMALLINT</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• 10 = INTEGER</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• 19 = BIGINT</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>For DECIMAL, it is the precision of the specified DECIMAL data type. 0 if the sequence is an alias.</td>
</tr>
<tr>
<td>ORIGIN</td>
<td>CHAR (1)</td>
<td></td>
<td>Origin of the sequence.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• S = System-generated sequence</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• U = User-generated sequence</td>
</tr>
<tr>
<td>REMARKS</td>
<td>VARCHAR (254)</td>
<td>Y</td>
<td>User-provided comments, or the null value.</td>
</tr>
</tbody>
</table>

Note:
1. The DEFINER column is included for backwards compatibility. See OWNER.
Each row represents a server-specific option value.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WRAPNAME</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>Name of the wrapper.</td>
</tr>
<tr>
<td>SERVERNAME</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>Uppercase name of the server.</td>
</tr>
<tr>
<td>SERVERTYPE</td>
<td>VARCHAR (30)</td>
<td>Y</td>
<td>Type of server.</td>
</tr>
<tr>
<td>SERVERVERSION</td>
<td>VARCHAR (18)</td>
<td>Y</td>
<td>Server version.</td>
</tr>
<tr>
<td>CREATE_TIME</td>
<td>TIMESTAMP</td>
<td></td>
<td>Time at which the entry was created.</td>
</tr>
<tr>
<td>OPTION</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Name of the server option.</td>
</tr>
<tr>
<td>SETTING</td>
<td>VARCHAR (2048)</td>
<td></td>
<td>Value of the server option.</td>
</tr>
<tr>
<td>SERVEROPTIONKEY</td>
<td>VARCHAR (18)</td>
<td></td>
<td>Uniquely identifies a row.</td>
</tr>
<tr>
<td>REMARKS</td>
<td>VARCHAR (254)</td>
<td>Y</td>
<td>User-provided comments, or the null value.</td>
</tr>
</tbody>
</table>
**SYSCAT.SERVERS**

Each row represents a data source.

*Table 168. SYSCAT.SERVERS Catalog View*

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WRAPNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Name of the wrapper.</td>
</tr>
<tr>
<td>SERVERNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Uppercase name of the server.</td>
</tr>
<tr>
<td>SERVERTYPE</td>
<td>VARCHAR (30)</td>
<td>Y</td>
<td>Type of server.</td>
</tr>
<tr>
<td>SERVERVERSION</td>
<td>VARCHAR (18)</td>
<td>Y</td>
<td>Server version.</td>
</tr>
<tr>
<td>REMARKS</td>
<td>VARCHAR (254)</td>
<td>Y</td>
<td>User-provided comments, or the null value.</td>
</tr>
</tbody>
</table>
Each row represents a service class.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SERVICECLASSNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Name of the service class.</td>
</tr>
<tr>
<td>PARENTSERVICECLASSNAME</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>Service class name of the parent service superclass.</td>
</tr>
<tr>
<td>SERVICECLASSID</td>
<td>SMALLINT</td>
<td></td>
<td>Identifier for the service class.</td>
</tr>
<tr>
<td>PARENTID</td>
<td>SMALLINT</td>
<td></td>
<td>Identifier for the parent service class for this service class. 0 if this service class is a super service class.</td>
</tr>
<tr>
<td>CREATE_TIME</td>
<td>TIMESTAMP</td>
<td></td>
<td>Time when the service class was created.</td>
</tr>
<tr>
<td>ALTER_TIME</td>
<td>TIMESTAMP</td>
<td></td>
<td>Time when the service class was last altered.</td>
</tr>
<tr>
<td>ENABLED</td>
<td>CHAR (1)</td>
<td></td>
<td>State of the service class.</td>
</tr>
<tr>
<td>AGENTPRIORITY</td>
<td>SMALLINT</td>
<td></td>
<td>Thread priority of the agents in the service class relative to the normal priority of DB2 threads.</td>
</tr>
<tr>
<td>PREFETCHPRIORITY</td>
<td>CHAR (1)</td>
<td></td>
<td>Prefetch priority of the agents in the service class.</td>
</tr>
<tr>
<td>BUFFERPOOLPRIORITY</td>
<td>CHAR (1)</td>
<td></td>
<td>Bufferpool priority of the agents in the service class.</td>
</tr>
<tr>
<td>INBOUNDCORRELATOR</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>String used to associate the service class with an operating system workload manager service class.</td>
</tr>
<tr>
<td>OUTBOUNDCORRELATOR</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>String used to associate the service class with an operating system workload manager service class.</td>
</tr>
<tr>
<td>COLLECTAGGACTDATA</td>
<td>CHAR (1)</td>
<td></td>
<td>Specifies what aggregate activity data should be captured for the service class by the applicable event monitor.</td>
</tr>
</tbody>
</table>

- **N** = Disabled
- **Y** = Enabled
- **-20** to **20** (Linux and UNIX)
- **-6** to **6** (Windows)
- **-32768** = not set
- **H** = High
- **L** = Low
- **M** = Medium
- **Blank** = not set
### Table 169. SYSCAT.SERVICECLASSES Catalog View (continued)

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>COLLECTAGGREQDATA</td>
<td>CHAR (1)</td>
<td></td>
<td>Specifies what aggregate activity data should be captured for the service class by the applicable event monitor.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• B = Collect base aggregate request data</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = None</td>
</tr>
<tr>
<td>COLLECTACTDATA</td>
<td>CHAR (1)</td>
<td></td>
<td>Specifies what activity data should be collected by the applicable event monitor.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• D = Activity data with details</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = None</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• S = Activity data with details and section environment</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• V = Activity data with details and values</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• W = Activity data without details</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• X = Activity data with details, section environment, and values</td>
</tr>
<tr>
<td>COLLECTACTPARTITION</td>
<td>CHAR (1)</td>
<td></td>
<td>Specifies where activity data is collected.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• C = Database partition of the coordinator of the activity</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• D = All database partitions</td>
</tr>
<tr>
<td>COLLECTREQMETRICS</td>
<td>CHAR (1)</td>
<td></td>
<td>Specifies the monitoring level for requests submitted by a connection that is associated with the service superclass.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• B = Collect base request metrics</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• E = Collect extended request metrics</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = None</td>
</tr>
<tr>
<td>REMARKS</td>
<td>VARCHAR (254)</td>
<td>Y</td>
<td>User-provided comments, or the null value.</td>
</tr>
</tbody>
</table>
SYSCAT.STATEMENTS

Each row represents an SQL statement in a package.

Table 170. SYSCAT.STATEMENTS Catalog View

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PKGSHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the package.</td>
</tr>
<tr>
<td>PKGNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the package.</td>
</tr>
<tr>
<td>STMTNO</td>
<td>INTEGER</td>
<td></td>
<td>Line number of the SQL statement in the source module of the application program.</td>
</tr>
<tr>
<td>SECTNO</td>
<td>SMALLINT</td>
<td></td>
<td>Number of the package section containing the SQL statement.</td>
</tr>
<tr>
<td>SEQNO</td>
<td>INTEGER</td>
<td></td>
<td>Always 1.</td>
</tr>
<tr>
<td>TEXT</td>
<td>CLOB (2M)</td>
<td></td>
<td>Text of the SQL statement.</td>
</tr>
<tr>
<td>UNIQUE_ID</td>
<td>CHAR (8) FOR BIT DATA</td>
<td></td>
<td>Identifier for a specific package when multiple packages having the same name exist.</td>
</tr>
<tr>
<td>VERSION</td>
<td>VARCHAR (64)</td>
<td>Y</td>
<td>Version identifier for the package.</td>
</tr>
</tbody>
</table>
SYSCAT.SURROGATEAUTHIDS

Each row represents a user or a group that has been granted SETSESSIONUSER privilege on a user or PUBLIC.

Table 171. SYSCAT.SURROGATEAUTHIDS Catalog View

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRANTOR</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Authorization ID that granted TRUSTEDID the ability to act as a surrogate. When the TRUSTEDID represents a trusted context object, this field represents the authorization ID that created or altered the trusted context object.</td>
</tr>
<tr>
<td>TRUSTEDID</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Identifier for the entity that is trusted to act as a surrogate.</td>
</tr>
</tbody>
</table>
| TRUSTEDIDTYPE     | CHAR (1)    |          | • C = Trusted context  
                          • G = Group  
                          • U = User                                                                                                    |
| SURROGATEAUTHID   | VARCHAR (128) |          | Surrogate authorization ID that can be assumed by TRUSTEDID. ‘PUBLIC’ indicates that TRUSTEDID can assume any authorization ID.     |
| SURROGATEAUTHIDTYPE | CHAR (1)   |          | • G = Group  
                          • U = User                                                                                                    |
| AUTHENTICATE      | CHAR (1)    |          | • N = No authentication is required  
                          • Y = Authentication token is required with the authorization ID to authenticate the user before the authorization ID can be assumed  
                          • Blank = TRUSTEDIDTYPE is not ‘C’                                                                 |
| CONTEXTROLE       | VARCHAR (128) | Y          | A specific role to be assigned to the assumed authorization ID, which supercedes the default role, if any, that is defined for the trusted context. Null value when TRUSTEDIDTYPE is not ‘C’. |
| GRANT_TIME        | TIMESTAMP   |          | Time at which the grant was made.                                                                                                          |
Each row represents a user, group, or role that has been granted one or more privileges on a table or view.

Table 172. SYSCAT.TABAUTH Catalog View

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRANTOR</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Grantor of the privilege.</td>
</tr>
<tr>
<td>GRANTORTYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>• S = Grantor is the system</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• U = Grantor is an individual user</td>
</tr>
<tr>
<td>GRANTEE</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Holder of the privilege.</td>
</tr>
<tr>
<td>GRANTEETYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>• G = Grantee is a group</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• R = Grantee is a role</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• U = Grantee is an individual user</td>
</tr>
<tr>
<td>TABSCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the table or view.</td>
</tr>
<tr>
<td>TABNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the table or view.</td>
</tr>
<tr>
<td>CONTROLAUTH</td>
<td>CHAR (1)</td>
<td></td>
<td>CONTROL privilege.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = Not held</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = Held but not grantable</td>
</tr>
<tr>
<td>ALTERAUTH</td>
<td>CHAR (1)</td>
<td></td>
<td>Privilege to alter the table; allow a parent</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>table to this table to drop its primary key or</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>unique constraint; allow a table to become a</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>materialized query table that references this</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>table or view in the materialized query; or</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>allow a table that references this table or</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>view in its materialized query to no longer</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>be a materialized query table.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• G = Held and grantable</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = Not held</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = Held</td>
</tr>
<tr>
<td>DELETEAUTH</td>
<td>CHAR (1)</td>
<td></td>
<td>Privilege to delete rows from a table or</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>updatable view.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• G = Held and grantable</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = Not held</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = Held</td>
</tr>
<tr>
<td>INDEXAUTH</td>
<td>CHAR (1)</td>
<td></td>
<td>Privilege to create an index on a table.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• G = Held and grantable</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = Not held</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = Held</td>
</tr>
<tr>
<td>INSERTAUTH</td>
<td>CHAR (1)</td>
<td></td>
<td>Privilege to insert rows into a table or</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>updatable view, or to run the import utility</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>against a table or view.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• G = Held and grantable</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = Not held</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = Held</td>
</tr>
</tbody>
</table>
### Table 172. SYSCAT.TABAUTH Catalog View (continued)

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>REFAUTH</td>
<td>CHAR (1)</td>
<td></td>
<td>Privilege to create and drop a foreign key referencing a table as the parent.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• G = Held and grantable</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = Not held</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = Held</td>
</tr>
<tr>
<td>SELECTAUTH</td>
<td>CHAR (1)</td>
<td></td>
<td>Privilege to retrieve rows from a table or view, create views on a table,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>or to run the export utility against a table or view.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• G = Held and grantable</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = Not held</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = Held</td>
</tr>
<tr>
<td>UPDATEAUTH</td>
<td>CHAR (1)</td>
<td></td>
<td>Privilege to run the UPDATE statement against a table or updatable view.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• G = Held and grantable</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = Not held</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = Held</td>
</tr>
</tbody>
</table>
Each row represents a table constraint of type CHECK, UNIQUE, PRIMARY KEY, or FOREIGN KEY. For table hierarchies, each constraint is recorded only at the level of the hierarchy where the constraint was created.

### Table 173. SYSCAT.TABCONST Catalog View

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONSTNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Name of the constraint.</td>
</tr>
<tr>
<td>TABSCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the table to which this constraint applies.</td>
</tr>
<tr>
<td>TABNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the table to which this constraint applies.</td>
</tr>
<tr>
<td>OWNER</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Authorization ID of the owner of the constraint.</td>
</tr>
<tr>
<td>OWNERTYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>• S = The owner is the system</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• U = The owner is an individual user</td>
</tr>
<tr>
<td>TYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>Indicates the constraint type.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• F = Foreign key</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• I = Functional dependency</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• K = Check</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• P = Primary key</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• U = Unique</td>
</tr>
<tr>
<td>ENFORCED</td>
<td>CHAR (1)</td>
<td></td>
<td>• N = Do not enforce constraint</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = Enforce constraint</td>
</tr>
<tr>
<td>CHECKEXISTINGDATA</td>
<td>CHAR (1)</td>
<td></td>
<td>• D = Defer checking any existing data</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• I = Immediately check existing data</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = Never check existing data</td>
</tr>
<tr>
<td>ENABLEQUERYOPT</td>
<td>CHAR (1)</td>
<td></td>
<td>• N = Query optimization is disabled</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = Query optimization is enabled</td>
</tr>
<tr>
<td>DEFINER</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Authorization ID of the owner of the constraint.</td>
</tr>
<tr>
<td>REMARKS</td>
<td>VARCHAR (254)</td>
<td>Y</td>
<td>User-provided comments, or the null value.</td>
</tr>
</tbody>
</table>

**Note:**
1. The DEFINER column is included for backwards compatibility. See OWNER.
Each row represents a dependency of a view or a materialized query table on some other object. The view or materialized query table depends on the object of type BTYPE of name BNAME, so a change to the object affects the view or materialized query table. Also encodes how privileges on views depend on privileges on underlying tables and views.

**Table 174. SYSCAT.TABDEP Catalog View**

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TABSCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the view or materialized query table.</td>
</tr>
<tr>
<td>TABNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the view or materialized query table.</td>
</tr>
<tr>
<td>DTYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>Type of the depending object.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• S = Materialized query table</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• T = Table (staging only)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• V = View (untyped)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• W = Typed view</td>
</tr>
<tr>
<td>OWNER</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Authorization ID of the creator of the view or materialized query table.</td>
</tr>
<tr>
<td>OWNERTYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>• U = The owner is an individual user</td>
</tr>
<tr>
<td>BTYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>Type of object on which there is a dependency. Possible values are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• A = Table alias</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• F = Routine</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• I = Index, if recording dependency on a base table</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• G = Global temporary table</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = Nickname</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• O = Privilege dependency on all subtables or subviews in a table or view hierarchy</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• R = User-defined structured type</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• S = Materialized query table</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• T = Table (untyped)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• U = Typed table</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• V = View (untyped)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• W = Typed view</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Z = XSR object</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• u = Module alias</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• v = Global variable</td>
</tr>
<tr>
<td>BSCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the object on which the view or materialized query table depends.</td>
</tr>
<tr>
<td>BMODULENAME</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>Unqualified name of the module to which the object on which there is a dependency belongs. The null value if not a module object.</td>
</tr>
<tr>
<td>BNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the object on which the view or materialized query table depends.</td>
</tr>
</tbody>
</table>
Table 174. SYSCAT.TABDEP Catalog View (continued)

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BMODULEID</td>
<td>INTEGER</td>
<td>Y</td>
<td>Identifier for the module of the object on which the view or materialized query table depends.</td>
</tr>
<tr>
<td>TABAUTH</td>
<td>SMALLINT</td>
<td>Y</td>
<td>If BTYPE is ‘N’, ‘O’, ‘S’, ‘T’, ‘U’, ‘V’, or ‘W’, encodes the privileges on the underlying table or view on which this view or materialized query table depends; the null value otherwise.</td>
</tr>
<tr>
<td>VARAUTH</td>
<td>SMALLINT</td>
<td>Y</td>
<td>If BTYPE is ‘v’, encodes the privileges on the underlying global variable on which this view or materialized query table depends; the null value otherwise.</td>
</tr>
<tr>
<td>DEFINER*</td>
<td>VARCHAR(128)</td>
<td></td>
<td>Authorization ID of the creator of the view or materialized query table.</td>
</tr>
</tbody>
</table>

Note:

1. The DEFINER column is included for backwards compatibility. See OWNER.
SYSCAT.TABDETACHEDDEP

Each row represents a detached dependency between a detached dependent table and a detached table.

Table 175. SYSCAT.TABDETACHEDDEP Catalog View

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TABSCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the detached table.</td>
</tr>
<tr>
<td>TABNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the detached table.</td>
</tr>
<tr>
<td>DEPTABSCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the detached dependent table.</td>
</tr>
<tr>
<td>DEPTABNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the detached dependent table.</td>
</tr>
</tbody>
</table>

Appendix D. Catalog views 913
SYSCAT.TABLES

Each row represents a table, view, alias, or nickname. Each table or view hierarchy has one additional row representing the hierarchy table or hierarchy view that implements the hierarchy. Catalog tables and views are included.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TABSCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the object</td>
</tr>
<tr>
<td>TABNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the object</td>
</tr>
<tr>
<td>OWNER</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Authorization ID of the owner of the table, view, alias, or nickname.</td>
</tr>
<tr>
<td>OWNERTYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>• S = The owner is the system</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• U = The owner is an individual user</td>
</tr>
<tr>
<td>TYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>Type of object.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• A = Alias</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• G = Created temporary table</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• H = Hierarchy table</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• L = Detached table</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = Nickname</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• S = Materialized query table</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• T = Table (untyped)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• U = Typed table</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• V = View (untyped)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• W = Typed view</td>
</tr>
<tr>
<td>STATUS</td>
<td>CHAR (1)</td>
<td></td>
<td>Status of the object.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• C = Set integrity pending</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = Normal</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• X = Inoperative</td>
</tr>
<tr>
<td>BASE_TABSCHEMA</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>If TYPE = ‘A’, contains the schema name of the table, view, alias, or nickname that is referenced by this alias; null value otherwise.</td>
</tr>
<tr>
<td>BASE_TABNAME</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>If TYPE = ‘A’, contains the unqualified name of the table, view, alias, or nickname that is referenced by this alias; null value otherwise.</td>
</tr>
<tr>
<td>ROWTYPESCHEMA</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>Schema name of the row type for this table, if applicable; null value otherwise.</td>
</tr>
<tr>
<td>ROWTYPENAME</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>Unqualified name of the row type for this table, if applicable; null value otherwise.</td>
</tr>
<tr>
<td>CREATE_TIME</td>
<td>TIMESTAMP</td>
<td></td>
<td>Time at which the object was created</td>
</tr>
<tr>
<td>ALTER_TIME</td>
<td>TIMESTAMP</td>
<td></td>
<td>Time at which the object was last altered</td>
</tr>
<tr>
<td>INVALIDATE_TIME</td>
<td>TIMESTAMP</td>
<td></td>
<td>Time at which the object was last invalidated</td>
</tr>
<tr>
<td>STATS_TIME</td>
<td>TIMESTAMP</td>
<td>Y</td>
<td>Time at which any change was last made to recorded statistics for this object. The null value if statistics are not collected.</td>
</tr>
<tr>
<td>COLCOUNT</td>
<td>SMALLINT</td>
<td></td>
<td>Number of columns, including inherited columns (if any).</td>
</tr>
<tr>
<td>TABLEID</td>
<td>SMALLINT</td>
<td></td>
<td>Internal logical object identifier</td>
</tr>
<tr>
<td>TBSPACEID</td>
<td>SMALLINT</td>
<td></td>
<td>Internal logical identifier for the primary table space for this object.</td>
</tr>
<tr>
<td>CARD</td>
<td>BIGINT</td>
<td></td>
<td>Total number of rows in the table; -1 if statistics are not collected.</td>
</tr>
<tr>
<td>Column Name</td>
<td>Data Type</td>
<td>Nullable</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>---------------</td>
<td>----------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>NPAGES</td>
<td>BIGINT</td>
<td></td>
<td>Total number of pages on which the rows of the table exist; -1 for a view or alias, or if statistics are not collected; -2 for a subtable or hierarchy table.</td>
</tr>
<tr>
<td>FPAGES</td>
<td>BIGINT</td>
<td></td>
<td>Total number of pages; -1 for a view or alias, or if statistics are not collected; -2 for a subtable or hierarchy table.</td>
</tr>
<tr>
<td>OVERFLOW</td>
<td>BIGINT</td>
<td></td>
<td>Total number of overflow records in the table; -1 for a view or alias, or if statistics are not collected; -2 for a subtable or hierarchy table.</td>
</tr>
<tr>
<td>TBSPACE</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>Name of the primary table space for the table. If no other table space is specified, all parts of the table are stored in this table space. The null value for aliases, views, and partitioned tables.</td>
</tr>
<tr>
<td>INDEX_TBSPACE</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>Name of the table space that holds all indexes created on this table. The null value for aliases, views, and partitioned tables, or if the INDEX IN clause was omitted or specified with the same value as the IN clause of the CREATE TABLE statement.</td>
</tr>
<tr>
<td>LONG_TBSPACE</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>Name of the table space that holds all long data (LONG or LOB column types) for this table. The null value for aliases, views, and partitioned tables, or if the LONG IN clause was omitted or specified with the same value as the IN clause of the CREATE TABLE statement.</td>
</tr>
<tr>
<td>PARENTS</td>
<td>SMALLINT</td>
<td>Y</td>
<td>Number of parent tables for this object; that is, the number of referential constraints in which this object is a dependent.</td>
</tr>
<tr>
<td>CHILDRENS</td>
<td>SMALLINT</td>
<td>Y</td>
<td>Number of dependent tables for this object; that is, the number of referential constraints in which this object is a parent.</td>
</tr>
<tr>
<td>SELFREFS</td>
<td>SMALLINT</td>
<td>Y</td>
<td>Number of self-referencing referential constraints for this object; that is, the number of referential constraints in which this object is both a parent and a dependent.</td>
</tr>
<tr>
<td>KEYCOLUMNS</td>
<td>SMALLINT</td>
<td>Y</td>
<td>Number of columns in the primary key.</td>
</tr>
<tr>
<td>KEYINDEXID</td>
<td>SMALLINT</td>
<td>Y</td>
<td>Index identifier for the primary key index; 0 or the null value if there is no primary key.</td>
</tr>
<tr>
<td>KEYUNIQUE</td>
<td>SMALLINT</td>
<td></td>
<td>Number of unique key constraints (other than the primary key constraint) defined on this object.</td>
</tr>
<tr>
<td>CHECKCOUNT</td>
<td>SMALLINT</td>
<td></td>
<td>Number of check constraints defined on this object.</td>
</tr>
<tr>
<td>DATACAPTURE</td>
<td>CHAR (1)</td>
<td></td>
<td>• L = Table participates in data replication, including replication of LONG VARCHAR and LONG VARGRAPHIC columns</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = Table does not participate in data replication</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = Table participates in data replication, excluding replication of LONG VARCHAR and LONG VARGRAPHIC columns</td>
</tr>
</tbody>
</table>
**SYSCAT.TABLES**

Table 176. **SYSCAT.TABLES** Catalog View (continued)

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONST_CHECKED</td>
<td>CHAR (32)</td>
<td></td>
<td>• Byte 1 represents foreign key constraint.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Byte 2 represents check constraint.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Byte 5 represents materialized query table.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Byte 6 represents generated column.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Byte 7 represents staging table.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Byte 8 represents data partitioning constraint.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Other bytes are reserved for future use.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Possible values are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• F = In byte 5, the materialized query table cannot be refreshed incrementally.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• In byte 7, the content of the staging table is incomplete and cannot be used for incremental refresh of the associated materialized query table.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = Not checked</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• U = Checked by user</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• W = Was in ‘U’ state when the table was placed in set integrity pending state</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = Checked by system</td>
</tr>
<tr>
<td>PMAP_ID</td>
<td>SMALLINT</td>
<td>Y</td>
<td>Identifier for the distribution map that is currently in use by this table (the null value for aliases or views).</td>
</tr>
<tr>
<td>PARTITION_MODE</td>
<td>CHAR (1)</td>
<td></td>
<td>Indicates how data is distributed among database partitions in a partitioned database system.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• H = Hashing</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• R = Replicated across database partitions</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Blank = No database partitioning</td>
</tr>
<tr>
<td>LOG_ATTRIBUTE</td>
<td>CHAR (1)</td>
<td></td>
<td>• Always 0. This column is no longer used.</td>
</tr>
<tr>
<td>PCTFREE</td>
<td>SMALLINT</td>
<td></td>
<td>Percentage of each page to be reserved for future inserts.</td>
</tr>
<tr>
<td>APPEND_MODE</td>
<td>CHAR (1)</td>
<td></td>
<td>Controls how rows are inserted into pages.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = New rows are inserted into existing spaces, if available</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = New rows are appended to the end of the data</td>
</tr>
<tr>
<td>REFRESH</td>
<td>CHAR (1)</td>
<td></td>
<td>Refresh mode.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• D = Deferred</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• I = Immediate</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• O = Once</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Blank = Not a materialized query table</td>
</tr>
<tr>
<td>REFRESH_TIME</td>
<td>TIMESTAMP</td>
<td>Y</td>
<td>For REFRESH = ‘D’ or ‘O’, time at which the data was last refreshed (REFRESH TABLE statement); null value otherwise.</td>
</tr>
<tr>
<td>LOCKSIZE</td>
<td>CHAR (1)</td>
<td></td>
<td>Indicates the preferred lock granularity for tables that are accessed by data manipulation language (DML) statements. Applies to tables only. Possible values are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• I = Block insert</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• R = Row</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• T = Table</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Blank = Not applicable</td>
</tr>
</tbody>
</table>
Table 176. SYSCAT.TABLES Catalog View (continued)

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VOLATILE</td>
<td>CHAR (1)</td>
<td></td>
<td>• C = Cardinality of the table is volatile</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Blank = Not applicable</td>
</tr>
<tr>
<td>ROW_FORMAT</td>
<td>CHAR (1)</td>
<td></td>
<td>Not used.</td>
</tr>
<tr>
<td>PROPERTY</td>
<td>VARCHAR (32)</td>
<td></td>
<td>Properties for a table. A single blank indicates that the table has no properties. The following is position within string, value, and meaning:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• 1, Y = User maintained materialized query table</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• 2, Y = Staging table</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• 3, Y = Propagate immediate</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• 11, Y = Nickname that will not be cached</td>
</tr>
<tr>
<td>STATISTICS_PROFILE</td>
<td>CLOB (10M)</td>
<td>Y</td>
<td>RUNSTATS command used to register a statistical profile for the object.</td>
</tr>
<tr>
<td>COMPRESSION</td>
<td>CHAR (1)</td>
<td></td>
<td>• B = Both value and row compression are activated</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = No compression is activated; a row format that does not support compression is used</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• R = Row compression is activated if licensed; a row format that supports compression might be used</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• V = Value compression is activated; a row format that supports compression is used</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Blank = Not applicable</td>
</tr>
<tr>
<td>ACCESS_MODE</td>
<td>CHAR (1)</td>
<td></td>
<td>Access restriction state of the object. These states only apply to objects that are in set integrity pending state or to objects that were processed by a SET INTEGRITY statement. Possible values are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• D = No data movement</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• F = Full access</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = No access</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• R = Read-only access</td>
</tr>
<tr>
<td>CLUSTERED</td>
<td>CHAR (1)</td>
<td>Y</td>
<td>• Y = Table is multidimensionally clustered (even if only by one dimension)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Null value = Table is not multidimensionally clustered</td>
</tr>
<tr>
<td>ACTIVE_BLOCKS</td>
<td>BIGINT</td>
<td></td>
<td>Total number of active blocks in the table, or -1. Applies to multidimensional clustering (MDC) tables only.</td>
</tr>
<tr>
<td>DROPRULE</td>
<td>CHAR (1)</td>
<td></td>
<td>• N = No rule</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• R = Restrict rule applies on drop</td>
</tr>
<tr>
<td>MAXFREESPACESHARE</td>
<td>SMALLINT</td>
<td></td>
<td>Reserved for future use.</td>
</tr>
<tr>
<td>AVERAGECOMPRESSEDROWSIZE</td>
<td>SMALLINT</td>
<td></td>
<td>Average length (in bytes) of compressed rows in this table; -1 if statistics are not collected.</td>
</tr>
<tr>
<td>AVERAGEROWCOMPRESSIONRATIO</td>
<td>REAL</td>
<td></td>
<td>For compressed rows in the table, this is the average compression ratio by row; that is, the average uncompressed row length divided by the average compressed row length; -1 if statistics are not collected.</td>
</tr>
<tr>
<td>AVERAGEROWSIZE</td>
<td>SMALLINT</td>
<td></td>
<td>Average length (in bytes) of both compressed and uncompressed rows in this table; -1 if statistics are not collected.</td>
</tr>
<tr>
<td>PERCENTCOMPRESSEDROWS</td>
<td>REAL</td>
<td></td>
<td>Compressed rows as a percentage of the total number of rows in the table; -1 if statistics are not collected.</td>
</tr>
</tbody>
</table>
### Table 176. SYSCAT.TABLES Catalog View (continued)

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOGINDEXBUILD</td>
<td>VARCHAR (3)</td>
<td>Y</td>
<td>Level of logging that is to be performed during create, recreate, or reorganize index operations on the table.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- OFF = Index build operations on the table will be logged minimally</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- ON = Index build operations on the table will be logged completely</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Null value = Value of the logindexbuild database configuration parameter will be used to determine whether or not index build operations are to be completely logged</td>
</tr>
<tr>
<td>CODEPAGE</td>
<td>SMALLINT</td>
<td></td>
<td>Code page of the object. This is the default code page used for all character columns, triggers, check constraints, and expression-generated columns.</td>
</tr>
<tr>
<td>COLLATIONSCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the collation for the table.</td>
</tr>
<tr>
<td>COLLATIONNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the collation for the table.</td>
</tr>
<tr>
<td>COLLATIONSCHEMA_ORDERBY</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the collation for ORDER BY clauses in the table.</td>
</tr>
<tr>
<td>COLLATIONNAME_ORDERBY</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the collation for ORDER BY clauses in the table.</td>
</tr>
<tr>
<td>ENCODING_SCHEME</td>
<td>CHAR (1)</td>
<td></td>
<td>- A = CCSID ASCII was specified</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- U = CCSID UNICODE was specified</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Blank = CCSID clause was not specified</td>
</tr>
<tr>
<td>PCTPAGESAVED</td>
<td>SMALLINT</td>
<td></td>
<td>Approximate percentage of pages saved in the table as a result of row compression. This value includes overhead bytes for each user data row in the table, but does not include the space that is consumed by dictionary overhead; -1 if statistics are not collected.</td>
</tr>
<tr>
<td>LAST_REGEN_TIME</td>
<td>TIMESTAMP</td>
<td>Y</td>
<td>Time at which any views or check constraints on the table were last regenerated.</td>
</tr>
<tr>
<td>SECPOLICYID</td>
<td>INTEGER</td>
<td></td>
<td>Identifier for the security policy protecting the table; 0 for non-protected tables.</td>
</tr>
<tr>
<td>PROTECTIONGRANULARITY</td>
<td>CHAR (1)</td>
<td></td>
<td>- B = Both column- and row-level granularity</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- C = Column-level granularity</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- R = Row-level granularity</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Blank = Non-protected table</td>
</tr>
<tr>
<td>AUDITPOLICYID</td>
<td>INTEGER</td>
<td>Y</td>
<td>Identifier for the audit policy.</td>
</tr>
<tr>
<td>AUDITPOLICYNAME</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>Name of the audit policy.</td>
</tr>
<tr>
<td>DEFINER1</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Authorization ID of the owner of the table, view, alias, or nickname.</td>
</tr>
<tr>
<td>ONCOMMIT</td>
<td>CHAR (1)</td>
<td></td>
<td>Specifies the action taken on the created temporary table when a COMMIT operation is performed.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- D = Delete rows</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- P = Preserve rows</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Blank = Table is not a created temporary table</td>
</tr>
<tr>
<td>LOGGED</td>
<td>CHAR (1)</td>
<td></td>
<td>Specifies whether the created temporary table is logged.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- N = Not logged</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Y = Logged</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Blank = Table is not a created temporary table</td>
</tr>
<tr>
<td>Column Name</td>
<td>Data Type</td>
<td>Nullable</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------</td>
<td>----------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| ONROLLBACK   | CHAR (1)    |          | Specifies the action taken on the created temporary table when a ROLLBACK operation is performed.  
|              |             |          | • D = Delete rows                                                           |
|              |             |          | • P = Preserve rows                                                          |
|              |             |          | • Blank = Table is not a created temporary table                           |
| LASTUSED     | DATE        |          | Date when the table was last used by any DML statement or the LOAD command. This column is not updated for an alias, created temporary table, nickname, or view. The default value is '0001-01-01'. This value is updated asynchronously. |
| REMARKS      | VARCHAR (254) | Y        | User-provided comments, or the null value.                                 |

**Note:**
1. The DEFINER column is included for backwards compatibility. See OWNER.
SYSCAT.TABLESPACES

Each row represents a table space.

Table 177. SYSCAT.TABLESPACES Catalog View

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TBSPACE</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Name of the table space.</td>
</tr>
<tr>
<td>OWNER</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Authorization ID of the owner of the table space.</td>
</tr>
<tr>
<td>OWNERTYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>- S = The owner is the system</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- U = The owner is an individual user</td>
</tr>
<tr>
<td>CREATE_TIME</td>
<td>TIMESTAMP</td>
<td></td>
<td>Time at which the table space was created.</td>
</tr>
<tr>
<td>TBSPACEID</td>
<td>INTEGER</td>
<td></td>
<td>Identifier for the table space.</td>
</tr>
<tr>
<td>TBSPACETYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>Type of table space.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- D = Database-managed space</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- S = System-managed space</td>
</tr>
<tr>
<td>DATATYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>Type of data that can be stored in this table space.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- A = All types of permanent data; regular table space</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- L = All types of permanent data; large table space</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- T = System temporary tables only</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- U = Created temporary tables or declared temporary tables only</td>
</tr>
<tr>
<td>EXTENTSIZE</td>
<td>INTEGER</td>
<td></td>
<td>Size of each extent, in pages of size PAGESIZE. This many pages are written</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>to one container in the table space before switching to the next container.</td>
</tr>
<tr>
<td>PREFETCHSIZE</td>
<td>INTEGER</td>
<td></td>
<td>Number of pages of size PAGESIZE to be read when prefetching is performed;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-1 when AUTOMATIC.</td>
</tr>
<tr>
<td>OVERHEAD</td>
<td>DOUBLE</td>
<td></td>
<td>Controller overhead and disk seek and latency time, in milliseconds (average</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>for the containers in this table space).</td>
</tr>
<tr>
<td>TRANSFERRATE</td>
<td>DOUBLE</td>
<td></td>
<td>Time to read one page of size PAGESIZE into the buffer (average for the con</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>tainers in this table space).</td>
</tr>
<tr>
<td>WRITEOVERHEAD</td>
<td>DOUBLE</td>
<td>Y</td>
<td>Reserved for future use.</td>
</tr>
<tr>
<td>WRITETRANSFERRATE</td>
<td>DOUBLE</td>
<td>Y</td>
<td>Time to write one page of size PAGESIZE from the buffer to the table space</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(average for the containers in this table space). The null value means the</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>same value as TRANSFERRATE will be used.</td>
</tr>
<tr>
<td>PAGESIZE</td>
<td>INTEGER</td>
<td></td>
<td>Size (in bytes) of pages in this table space.</td>
</tr>
<tr>
<td>DBPGNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Name of the database partition group that is associated with this table sp</td>
</tr>
<tr>
<td>BUFFERPOOLID</td>
<td>INTEGER</td>
<td></td>
<td>Identifier for the buffer pool that is used by this table space (1 indicat</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>es the default buffer pool).</td>
</tr>
</tbody>
</table>
### SYSCAT.TABLESPACES

**Table 177. SYSCAT.TABLESPACES Catalog View (continued)**

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
</table>
| DROP_RECOVERY     | CHAR (1)    |          | Indicates whether or not tables in this table space can be recovered after a drop table operation.  
|                   |             |          | • N = Tables are not recoverable  
|                   |             |          | • Y = Tables are recoverable                                                  |
| NGNAME\(^1\)     | VARCHAR (128)|          | Name of the database partition group that is associated with this table space. |
| DEFINER\(^2\)    | VARCHAR (128)|          | Authorization ID of the owner of the table space.                             |
| DATAPRIORITY      | CHAR (1)    |          | Reserved for future use.                                                     |
| REMARKS           | VARCHAR (254)| Y        | User-provided comments, or the null value.                                   |

**Note:**
1. The NGNAME column is included for backwards compatibility. See DBPGNAME.
2. The DEFINER column is included for backwards compatibility. See OWNER.
Each row represents an option that is associated with a remote table.

Table 178. SYSCAT.TABOPTIONS Catalog View

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TABSCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of a table, view, alias, or nickname.</td>
</tr>
<tr>
<td>TABNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of a table, view, alias, or nickname.</td>
</tr>
<tr>
<td>OPTION</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Name of the table option.</td>
</tr>
<tr>
<td>SETTING</td>
<td>CLOB (32K)</td>
<td></td>
<td>Value of the table option.</td>
</tr>
</tbody>
</table>
Each row represents a user, group, or role that has been granted the USE privilege on a particular table space in the database.

Table 179. SYSCAT.TBSPACEAUTH Catalog View

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRANTOR</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Grantor of the privilege.</td>
</tr>
<tr>
<td>GRANTORTYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>• S = Grantor is the system</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• U = Grantor is an individual user</td>
</tr>
<tr>
<td>GRANTEE</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Holder of the privilege.</td>
</tr>
<tr>
<td>GRANTEETYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>• G = Grantee is a group</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• R = Grantee is a role</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• U = Grantee is a individual user</td>
</tr>
<tr>
<td>TBSPACE</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Name of the table space.</td>
</tr>
<tr>
<td>USEAUTH</td>
<td>CHAR (1)</td>
<td></td>
<td>Privilege to create tables within the table space.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• G = Held and grantable</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = Not held</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = Held</td>
</tr>
</tbody>
</table>
### SYSCAT.THRESHOLDS

Each row represents a threshold.

**Table 180. SYSCAT.THRESHOLDS Catalog View**

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>THRESHOLDNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Name of the threshold.</td>
</tr>
<tr>
<td>THRESHOLDID</td>
<td>INTEGER</td>
<td></td>
<td>Identifier for the threshold.</td>
</tr>
<tr>
<td>ORIGIN</td>
<td>CHAR (1)</td>
<td></td>
<td>Origin of the threshold.</td>
</tr>
<tr>
<td></td>
<td>U = Threshold was created by a user</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>W = Threshold was created through a work action set</td>
<td></td>
<td></td>
</tr>
<tr>
<td>THRESHOLDCLASS</td>
<td>CHAR (1)</td>
<td></td>
<td>Classification of the threshold.</td>
</tr>
<tr>
<td></td>
<td>A = Aggregate threshold</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>C = Activity threshold</td>
<td></td>
<td></td>
</tr>
<tr>
<td>THRESHOLDPREDICATE</td>
<td>VARCHAR (15)</td>
<td></td>
<td>Type of the threshold. Possible values are:</td>
</tr>
<tr>
<td></td>
<td>AGGTEMPSPACE</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CONCDBC</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CONCWCN</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CONCWOC</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CONNIDLETIME</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CPU TIME</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CPU TIME IN SC</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>DBCCONN</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>EST SQL COST</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ROWS READ</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ROWS READ IN SC</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ROWS RET</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SCONN</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>TEMPSPACE</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>TOTAL TIME</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>UOW TOTAL TIME</td>
<td></td>
<td></td>
</tr>
<tr>
<td>THRESHOLDPREDICATEID</td>
<td>SMALLINT</td>
<td></td>
<td>Identifier for the threshold predicate.</td>
</tr>
<tr>
<td>DOMAIN</td>
<td>CHAR (2)</td>
<td></td>
<td>Domain of the threshold.</td>
</tr>
<tr>
<td></td>
<td>DB = Database</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SB = Service subclass</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SP = Service superclass</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>WA = Work action set</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>WD = Workload definition</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DOMAINID</td>
<td>INTEGER</td>
<td></td>
<td>Identifier for the object with which the threshold is associated. This can be a service class, work action or workload unique ID. If this is a database threshold, this value is 0.</td>
</tr>
<tr>
<td>ENFORCEMENT</td>
<td>CHAR (1)</td>
<td></td>
<td>Scope of enforcement for the threshold.</td>
</tr>
<tr>
<td></td>
<td>D = Database</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>P = Database partition</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>W = Workload occurrence</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Column Name</td>
<td>Data Type</td>
<td>Nullable</td>
<td>Description</td>
</tr>
<tr>
<td>------------------</td>
<td>----------------</td>
<td>----------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>QUEUING</td>
<td>CHAR (1)</td>
<td></td>
<td>• N = The threshold is not queueing</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = The threshold is queueing</td>
</tr>
<tr>
<td>MAXVALUE</td>
<td>BIGINT</td>
<td></td>
<td>Upper bound specified by the threshold.</td>
</tr>
<tr>
<td>QUEUESIZE</td>
<td>INTEGER</td>
<td></td>
<td>If QUEUEING is 'Y', the size of the queue. -1 otherwise.</td>
</tr>
<tr>
<td>COLLECTACTDATA</td>
<td>CHAR (1)</td>
<td></td>
<td>Specifies what activity data should be collected by the applicable event monitor.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• D = Activity data with details</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = None</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• S = Activity data with details and section environment</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• V = Activity data with details and values</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• W = Activity data without details</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• X = Activity data with details, section environment, and values</td>
</tr>
<tr>
<td>COLLECTACTPARTITION</td>
<td>CHAR (1)</td>
<td></td>
<td>Specifies where activity data is collected.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• C = Database partition of the coordinator of the activity</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• D = All database partitions</td>
</tr>
<tr>
<td>EXECUTION</td>
<td>CHAR (1)</td>
<td></td>
<td>Indicates the execution action taken after a threshold has been exceeded.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• C = Execution continues</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• F = Application is forced off the system</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• R = Execution is remapped to a different service subclass</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• S = Execution stops</td>
</tr>
<tr>
<td>REMAPSCID</td>
<td>SMALLINT</td>
<td></td>
<td>Target service subclass ID of the REMAP ACTIVITY action.</td>
</tr>
<tr>
<td>VIOLATIONRECORDLOGGED</td>
<td>CHAR (1)</td>
<td></td>
<td>Indicates whether a record is written to the event monitor upon threshold violation.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = No</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = Yes</td>
</tr>
<tr>
<td>CHECKINTERVAL</td>
<td>INTEGER</td>
<td></td>
<td>The interval, in seconds, in which the threshold condition is checked if THRESHOLDPREDICATE is:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• ‘CPUTIME’</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• ‘CPUTIMEINSC’</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• ‘ROWSREAD’</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• ‘ROWSREADINSC’</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Otherwise, -1.</td>
</tr>
<tr>
<td>ENABLED</td>
<td>CHAR (1)</td>
<td></td>
<td>• N = This threshold is disabled</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = This threshold is enabled</td>
</tr>
<tr>
<td>CREATE_TIME</td>
<td>TIMESTAMP</td>
<td></td>
<td>Time at which the threshold was created.</td>
</tr>
<tr>
<td>ALTER_TIME</td>
<td>TIMESTAMP</td>
<td></td>
<td>Time at which the threshold was last altered.</td>
</tr>
<tr>
<td>REMARKS</td>
<td>VARCHAR (254)</td>
<td>Y</td>
<td>User-provided comments, or the null value.</td>
</tr>
</tbody>
</table>
SYSCAT.TRANSFORMS

Each row represents the functions that handle transformations between a user-defined type and a base SQL type, or the reverse.

### Table 181. SYSCAT.TRANSFORMS Catalog View

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TYPEID</td>
<td>SMALLINT</td>
<td></td>
<td>Identifier for the data type.</td>
</tr>
<tr>
<td>TYPESCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the data type if TYPEMODULEID is null; otherwise schema name of the module to which the data type belongs.</td>
</tr>
<tr>
<td>TYPENAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the data type.</td>
</tr>
<tr>
<td>GROUPNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Name of the transform group.</td>
</tr>
<tr>
<td>FUNCID</td>
<td>INTEGER</td>
<td></td>
<td>Identifier for the routine.</td>
</tr>
<tr>
<td>FUNCSHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the routine if ROUTINEMODULEID is null; otherwise schema name of the module to which the routine belongs.</td>
</tr>
<tr>
<td>FUNCNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the routine.</td>
</tr>
<tr>
<td>SPECIFICNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Name of the routine instance (might be system-generated).</td>
</tr>
<tr>
<td>TRANSFORMTYPE</td>
<td>VARCHAR(8)</td>
<td></td>
<td>• 'FROM SQL' = Transform function transforms a structured type from SQL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• 'TO SQL' = Transform function transforms a structured type to SQL</td>
</tr>
<tr>
<td>FORMAT</td>
<td>CHAR (1)</td>
<td></td>
<td>Format produced by the FROM SQL transform.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• S = Structured data type</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• U = User-defined</td>
</tr>
<tr>
<td>MAXLENGTH</td>
<td>INTEGER</td>
<td>Y</td>
<td>Maximum length (in bytes) of output from the FROM SQL transform; the null value for TO SQL transforms.</td>
</tr>
<tr>
<td>ORIGIN</td>
<td>CHAR (1)</td>
<td></td>
<td>Source of this group of transforms.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• O = Original transform group (built-in or system-defined)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• R = Redefined transform group (only built-in groups can be redefined)</td>
</tr>
<tr>
<td>REMARKS</td>
<td>VARCHAR (254)</td>
<td>Y</td>
<td>User-provided comments, or the null value.</td>
</tr>
</tbody>
</table>
SYSCAT.TRIGDEP

Each row represents a dependency of a trigger on some other object. The trigger depends on the object of type BTYPE of name BNAME, so a change to the object affects the trigger.

Table 182. SYSCAT.TRIGDEP Catalog View

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRIGSCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the trigger.</td>
</tr>
<tr>
<td>TRIGNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the trigger.</td>
</tr>
<tr>
<td>BTYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>Type of object on which there is a dependency. Possible values are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• A = Table alias</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• B = Trigger</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• F = Routine</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• G = Global temporary table</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• H = Hierarchy table</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• K = Package</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• L = Detached table</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = Nickname</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• O = Privilege dependency on all subtables or subviews in a table or view hierarchy</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Q = Sequence</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• R = User-defined data type</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• S = Materialized query table</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• T = Table (not typed)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• U = Typed table</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• V = View (not typed)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• W = Typed view</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• X = Index extension</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Z = XSR object</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• q = Sequence alias</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• u = Module alias</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• v = Global variable</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• * = Anchored to the row of a base table</td>
</tr>
<tr>
<td>BSCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the object on which there is a dependency.</td>
</tr>
<tr>
<td>BMODULENAME</td>
<td>VARCHAR(128)</td>
<td>Y</td>
<td>Unqualified name of the module to which the object on which a dependency belongs. The null value if not a module object.</td>
</tr>
<tr>
<td>BNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the object on which there is a dependency. For routines (BTYPE = 'F'), this is the specific name.</td>
</tr>
<tr>
<td>BMODULEID</td>
<td>INTEGER</td>
<td>Y</td>
<td>Identifier for the module of the object on which there is a dependency.</td>
</tr>
<tr>
<td>TABAUTH</td>
<td>SMALLINT</td>
<td>Y</td>
<td>If BTYPE = 'O', 'S', 'T', 'U', 'V', 'W', or 'v', encodes the privileges on the table or view that are required by a dependent trigger; null value otherwise.</td>
</tr>
</tbody>
</table>

Appendix D. Catalog views 927
Each row represents a trigger. For table hierarchies, each trigger is recorded only at the level of the hierarchy where the trigger was created.

**Table 183. SYSCAT.TRIGGERS Catalog View**

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRIGSCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the trigger.</td>
</tr>
<tr>
<td>TRIGNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the trigger.</td>
</tr>
<tr>
<td>OWNER</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Authorization ID of the owner of the trigger.</td>
</tr>
<tr>
<td>OWNER_TYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>• S = The owner is the system</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• U = The owner is an individual user</td>
</tr>
<tr>
<td>TABSCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the table or view to which this trigger applies.</td>
</tr>
<tr>
<td>TABNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the table or view to which this trigger applies.</td>
</tr>
<tr>
<td>TRIGTIME</td>
<td>CHAR (1)</td>
<td></td>
<td>Time at which triggered actions are applied to the base table, relative to the</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>event that fired the trigger.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• A = Trigger is applied after the event</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• B = Trigger is applied before the event</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• I = Trigger is applied instead of the event</td>
</tr>
<tr>
<td>TRIGEVENT</td>
<td>CHAR (1)</td>
<td></td>
<td>Event that fires the trigger.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• D = Delete operation</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• I = Insert operation</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• U = Update operation</td>
</tr>
<tr>
<td>GRANULARITY</td>
<td>CHAR (1)</td>
<td></td>
<td>Trigger is executed once per:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• R = Row</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• S = Statement</td>
</tr>
<tr>
<td>VALID</td>
<td>CHAR (1)</td>
<td></td>
<td>• N = Trigger is invalid</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• X = Trigger is inoperative and must be recreated</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = Trigger is valid</td>
</tr>
<tr>
<td>CREATE_TIME</td>
<td>TIMESTAMP</td>
<td></td>
<td>Time at which the trigger was defined. Used in resolving functions and types.</td>
</tr>
<tr>
<td>QUALIFIER</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Value of the default schema at the time of object definition. Used to complete</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>any unqualified references.</td>
</tr>
<tr>
<td>FUNC_PATH</td>
<td>CLOB (2K)</td>
<td></td>
<td>SQL path in effect when the trigger was defined.</td>
</tr>
<tr>
<td>TEXT</td>
<td>CLOB (2M)</td>
<td></td>
<td>Full text of the CREATE TRIGGER statement, exactly as typed.</td>
</tr>
<tr>
<td>LAST_REGEN_TIME</td>
<td>TIMESTAMP</td>
<td></td>
<td>Time at which the packed descriptor for the trigger was last regenerated.</td>
</tr>
<tr>
<td>COLLATIONSHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the collation for the trigger.</td>
</tr>
<tr>
<td>COLLATIONNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the collation for the trigger.</td>
</tr>
<tr>
<td>COLLATIONSHEMA_ORDERBY</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the collation for ORDER BY clauses in the trigger.</td>
</tr>
<tr>
<td>COLLATIONNAME_ORDERBY</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the collation for ORDER BY clauses in the trigger.</td>
</tr>
<tr>
<td>PRECOMPILE_OPTIONS</td>
<td>VARCHAR (1024)</td>
<td>Y</td>
<td>The precompile and bind options that were in effect when the compiled trigger</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>was created. The null value if the trigger is not compiled.</td>
</tr>
</tbody>
</table>
### Table 183. SYSCAT.TRIGGERS Catalog View (continued)

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMPIL_OPTIONS</td>
<td>VARCHAR (1024)</td>
<td>Y</td>
<td>The value of the SQL_CCFLAGS special register that was in effect when the compiled trigger was created and inquiry directives were present. An empty string if no inquiry directives were present in the compiled trigger. The null value if the trigger is not compiled.</td>
</tr>
<tr>
<td>DEFINER(^1)</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Authorization ID of the owner of the trigger.</td>
</tr>
<tr>
<td>REMARKS</td>
<td>VARCHAR (254)</td>
<td>Y</td>
<td>User-provided comments, or the null value.</td>
</tr>
</tbody>
</table>

**Note:**

1. The DEFINER column is included for backwards compatibility. See OWNER.
SYSCAT.TYPEMAPPINGS

Each row represents a data type mapping between a locally-defined data type and a data source data type. There are two mapping types (mapping directions):

- Forward type mappings map a data source data type to a locally-defined data type.
- Reverse type mappings map a locally-defined data type to a data source data type.

Table 184. SYSCAT.TYPEMAPPINGS Catalog View

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TYPE_MAPPING</td>
<td>VARCHAR (18)</td>
<td></td>
<td>Name of the type mapping (might be system-generated).</td>
</tr>
<tr>
<td>MAPPINGDIRECTION</td>
<td>CHAR (1)</td>
<td></td>
<td>Indicates whether this type mapping is a forward or a reverse type mapping.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>F = Forward type mapping</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>R = Reverse type mapping</td>
<td></td>
</tr>
<tr>
<td>TYPE_SCHEMA</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>Schema name of the local type in a data type mapping; the null value for built-in types.</td>
</tr>
<tr>
<td>TYPENAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the local type in a data type mapping.</td>
</tr>
<tr>
<td>TYPEID</td>
<td>SMALLINT</td>
<td></td>
<td>Identifier for the data type.</td>
</tr>
<tr>
<td>SOURCE_TYPEID</td>
<td>SMALLINT</td>
<td></td>
<td>Identifier for the source type.</td>
</tr>
<tr>
<td>OWNER</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Authorization ID of the owner of the type mapping. ‘SYSIBM’ indicates a built-in type mapping.</td>
</tr>
<tr>
<td>OWNERTYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>• S = The owner is the system</td>
</tr>
<tr>
<td></td>
<td></td>
<td>U = The owner is an individual user</td>
<td></td>
</tr>
<tr>
<td>LENGTH</td>
<td>INTEGER</td>
<td>Y</td>
<td>Maximum length or precision of the local data type in this mapping. If the null value, the system determines the maximum length or precision. For character types, represents the maximum number of bytes.</td>
</tr>
<tr>
<td>SCALE</td>
<td>SMALLINT</td>
<td>Y</td>
<td>Maximum number of digits in the fractional part of a local decimal value or the maximum number of digits of fractional seconds of a local TIMESTAMP value in this mapping. If the null value, the system determines the maximum number.</td>
</tr>
<tr>
<td>LOWER_LEN</td>
<td>INTEGER</td>
<td>Y</td>
<td>Minimum length or precision of the local data type in this mapping. If the null value, the system determines the minimum length or precision. For character types, represents the minimum number of bytes.</td>
</tr>
<tr>
<td>UPPER_LEN</td>
<td>INTEGER</td>
<td>Y</td>
<td>Maximum length or precision of the local data type in this mapping. If the null value, the system determines the maximum length or precision. For character types, represents the maximum number of bytes.</td>
</tr>
</tbody>
</table>
Table 184. **SYSCAT.TYPMAPPINGS** Catalog View (continued)

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOWER_SCALE</td>
<td>SMALLINT</td>
<td>Y</td>
<td>Minimum number of digits in the fractional part of a local decimal value or the minimum number of digits of fractional seconds of a local TIMESTAMP value in this mapping. If the null value, the system determines the minimum number.</td>
</tr>
<tr>
<td>UPPER_SCALE</td>
<td>SMALLINT</td>
<td>Y</td>
<td>Maximum number of digits in the fractional part of a local decimal value or the maximum number of digits of fractional seconds of a local TIMESTAMP value in this mapping. If the null value, the system determines the maximum number.</td>
</tr>
<tr>
<td>S_OPR_P</td>
<td>CHAR (2)</td>
<td>Y</td>
<td>Relationship between the scale and precision of a local decimal value in this mapping. Basic comparison operators (=, &lt;, &gt;, &lt;=, &gt;=, &lt;&gt; ) can be used. A null value indicates that no specific relationship is required.</td>
</tr>
<tr>
<td>BIT_DATA</td>
<td>CHAR (1)</td>
<td>Y</td>
<td>Indicates whether or not this character type is for bit data. Possible values are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = This type is not for bit data</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = This type is for bit data</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Null value = This is not a character data type, or the system determines the bit data attribute</td>
</tr>
<tr>
<td>WRAPNAME</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>Data access protocol (wrapper) to which this mapping applies.</td>
</tr>
<tr>
<td>SERVERNAME</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>Uppercase name of the server.</td>
</tr>
<tr>
<td>SERVERTYPE</td>
<td>VARCHAR (30)</td>
<td>Y</td>
<td>Type of server.</td>
</tr>
<tr>
<td>SERVERVERSION</td>
<td>VARCHAR (18)</td>
<td>Y</td>
<td>Server version.</td>
</tr>
<tr>
<td>REMOTE_TYPESCHEMA</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>Schema name of the data source data type.</td>
</tr>
<tr>
<td>REMOTE_TYPENAME</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>Unqualified name of the data source data type.</td>
</tr>
<tr>
<td>REMOTE_META_TYPE</td>
<td>CHAR (1)</td>
<td>Y</td>
<td>Indicates whether this remote type is a system built-in type or a distinct type.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• S = System built-in type</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• T = Distinct type</td>
</tr>
<tr>
<td>REMOTE_LOWER_LEN</td>
<td>INTEGER</td>
<td>Y</td>
<td>Minimum length or precision of the remote data type in this mapping, or the null value. For character types, represents the minimum number of characters (not bytes). For binary types, represents the minimum number of bytes. A value of -1 indicates that the default length or precision is used, or that the remote type does not have a length or precision.</td>
</tr>
</tbody>
</table>
Table 184. SYSCAT.TYPEMAPPINGS Catalog View (continued)

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>REMOTE_UPPER_LEN</td>
<td>INTEGER</td>
<td>Y</td>
<td>Maximum length or precision of the remote data type in this mapping, or the null value. For character types, represents the maximum number of characters (not bytes). For binary types, represents the maximum number of bytes. A value of -1 indicates that the default length or precision is used, or that the remote type does not have a length or precision.</td>
</tr>
<tr>
<td>REMOTE_LOWERSCALE</td>
<td>SMALLINT</td>
<td>Y</td>
<td>Minimum number of digits in the fractional part of a remote decimal value or the minimum number of digits of fractional seconds of a remote TIMESTAMP value in this mapping, or the null value.</td>
</tr>
<tr>
<td>REMOTE_UPPERSCALE</td>
<td>SMALLINT</td>
<td>Y</td>
<td>Maximum number of digits in the fractional part of a remote decimal value or the maximum number of digits of fractional seconds of a remote TIMESTAMP value in this mapping, or the null value.</td>
</tr>
<tr>
<td>REMOTE_S_OPR_P</td>
<td>CHAR (2)</td>
<td>Y</td>
<td>Relationship between the scale and precision of a remote decimal value in this mapping. Basic comparison operators (=, &lt;, &gt;, &lt;=, &gt;=, &lt;&gt;) can be used. A null value indicates that no specific relationship is required.</td>
</tr>
</tbody>
</table>
| REMOTE_BIT_DATA  | CHAR (1)     | Y        | Indicates whether or not this remote character type is for bit data. Possible values are:  
|                  |              |          | • N = This type is not for bit data  
|                  |              |          | • Y = This type is for bit data  
|                  |              |          | • Null value = This is not a character data type, or the system determines the bit data attribute |
| USER_DEFINED     | CHAR (1)     |          | Indicates whether or not the mapping is user-defined. The value is always ‘Y’; that is, the mapping is always user-defined. |
| CREATE_TIME      | TIMESTAMP    |          | Time at which this mapping was created. |
| DEFINER         | VARCHAR (128)|          | Authorization ID of the owner of the type mapping. ‘SYSIBM’ indicates a built-in type mapping. |
| REMARKS          | VARCHAR (254)| Y        | User-provided comments, or the null value. |

Note:

1. The DEFINER column is included for backwards compatibility. See OWNER.
SYSCAT.USEROPTIONS

Each row represents a server-specific user option value.

Table 185. SYSCAT.USEROPTIONS Catalog View

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AUTHID</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Local authorization ID, in uppercase characters.</td>
</tr>
<tr>
<td>AUTHIDTYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>• U = Grantee is an individual user</td>
</tr>
<tr>
<td>SERVERNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Name of the server on which the user is defined</td>
</tr>
<tr>
<td>OPTION</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Name of the user option.</td>
</tr>
<tr>
<td>SETTING</td>
<td>VARCHAR (2048)</td>
<td></td>
<td>Value of the user option.</td>
</tr>
</tbody>
</table>
Each row represents a user, group, or role that has been granted one or more privileges by a specific grantor on a global variable in the database that is not defined in a module.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRANTOR</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Grantor of the privilege.</td>
</tr>
<tr>
<td>GRANTORTYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>• S = Grantor is the system</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• U = Grantor is an individual user</td>
</tr>
<tr>
<td>GRANTEE</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Holder of the privilege.</td>
</tr>
<tr>
<td>GRANTEETYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>• G = Grantee is a group</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• R = Grantee is a role</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• U = Grantee is an individual user</td>
</tr>
<tr>
<td>VARSHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the global variable if VARMODULEID is null; otherwise schema</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>name of the module to which the global variable belongs.</td>
</tr>
<tr>
<td>VARNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the global variable.</td>
</tr>
<tr>
<td>VARID</td>
<td>INTEGER</td>
<td></td>
<td>Identifier for the global variable.</td>
</tr>
<tr>
<td>READAUTH</td>
<td>CHAR (1)</td>
<td></td>
<td>Privilege to read the global variable.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• G = Held and grantable</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = Not held</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = Held</td>
</tr>
<tr>
<td>WRITEAUTH</td>
<td>CHAR (1)</td>
<td></td>
<td>Privilege to write the global variable.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• G = Held and grantable</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = Not held</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = Held</td>
</tr>
</tbody>
</table>
Each row represents a dependency of a global variable on some other object. The global variable depends on the object of type BTYPE of name BNAME, so a change to the object affects the global variable.

Table 187. SYSCAT.VARIABLEDEP Catalog View

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VARSHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the global variable that has dependencies on another object.</td>
</tr>
<tr>
<td>VARMODULENAME</td>
<td>VARCHAR(128)</td>
<td>Y</td>
<td>Unqualified name of the module to which the global variable belongs. The null value if not a module variable.</td>
</tr>
<tr>
<td>VARNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the global variable that has dependencies on another object.</td>
</tr>
<tr>
<td>VARMODULEID</td>
<td>INTEGER</td>
<td>Y</td>
<td>Identifier for the module of the object that has dependencies on another object.</td>
</tr>
<tr>
<td>BTYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>Type of object on which there is a dependency. Possible values are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• A = Table alias</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• F = Routine</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• G = Global temporary table</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• H = Hierarchy table</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = Nickname</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• O = Privilege dependency on all subtables or subviews in a table or view hierarchy</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• R = User-defined data type</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• S = Materialized query table</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• T = Table (not typed)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• U = Typed table</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• V = View (not typed)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• W = Typed view</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• q = Sequence alias</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• u = Module alias</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• v = Global variable</td>
</tr>
<tr>
<td>BSCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the object on which there is a dependency.</td>
</tr>
<tr>
<td>BMODULENAME</td>
<td>VARCHAR(128)</td>
<td>Y</td>
<td>Unqualified name of the module to which the object on which a dependency belongs. The null value if not a module object.</td>
</tr>
<tr>
<td>BNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the object on which there is a dependency. For routines (BTYPE = 'F'), this is the specific name.</td>
</tr>
<tr>
<td>BMODULEID</td>
<td>INTEGER</td>
<td>Y</td>
<td>Identifier for the module of the object on which there is a dependency.</td>
</tr>
<tr>
<td>TABAUTH</td>
<td>SMALLINT</td>
<td>Y</td>
<td>If BTYPE = ‘O’, ‘S’, ‘T’, ‘U’, ‘V’, ‘W’, or ‘v’, encodes the privileges on the table or view that are required by the dependent global variable; the null value otherwise.</td>
</tr>
</tbody>
</table>
### SYSCAT.VARIABLES

Each row represents a global variable.

**Table 188. SYSCAT.VARIABLES Catalog View**

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VARSHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the global variable if VARMODULEID is null; otherwise schema name of the module to which the global variable belongs.</td>
</tr>
<tr>
<td>VARMODULENAME</td>
<td>VARCHAR(128)</td>
<td>Y</td>
<td>Unqualified name of the module to which the global variable belongs. The null value if not a module variable.</td>
</tr>
<tr>
<td>VARNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the global variable.</td>
</tr>
<tr>
<td>VARMODULEID</td>
<td>INTEGER</td>
<td>Y</td>
<td>Identifier for the module to which the global variable belongs. The null value if not a module variable.</td>
</tr>
<tr>
<td>VARID</td>
<td>INTEGER</td>
<td></td>
<td>Identifier for the global variable.</td>
</tr>
<tr>
<td>OWNER</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Authorization ID of the owner of the global variable.</td>
</tr>
<tr>
<td>OWNERTYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>• U = The owner is an individual user</td>
</tr>
<tr>
<td>CREATE_TIME</td>
<td>TIMESTAMP</td>
<td></td>
<td>Time at which the global variable was created.</td>
</tr>
<tr>
<td>LAST_REGEN_TIME</td>
<td>TIMESTAMP</td>
<td></td>
<td>Time at which the default expression was last regenerated.</td>
</tr>
<tr>
<td>VALID</td>
<td>CHAR (1)</td>
<td></td>
<td>• N = The global variable is invalid</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = The global variable is valid</td>
</tr>
<tr>
<td>PUBLISHED</td>
<td>CHAR (1)</td>
<td></td>
<td>Indicates whether the module variable can be referenced outside its module.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = The module variable is not published</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = The module variable is published</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Blank = Not applicable</td>
</tr>
<tr>
<td>TYPESCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the data type if TYPEMODULEID is null; otherwise schema name of the module to which the data type belongs.</td>
</tr>
<tr>
<td>TYPEMODULENAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the module to which the variable data type belongs. The null value if the variable data type does not belong to a module.</td>
</tr>
<tr>
<td>TYPENAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the data type.</td>
</tr>
<tr>
<td>TYPEMODULEID</td>
<td>INTEGER</td>
<td>Y</td>
<td>Identifier for the module to which the variable data type belongs. The null value if the variable data type does not belong to a module.</td>
</tr>
<tr>
<td>LENGTH</td>
<td>INTEGER</td>
<td></td>
<td>Maximum length of the global variable.</td>
</tr>
<tr>
<td>SCALE</td>
<td>SMALLINT</td>
<td></td>
<td>Scale if the global variable data type is DECIMAL or distinct type based on DECIMAL; the number of digits of fractional seconds if the global variable data type is TIMESTAMP or distinct type based on TIMESTAMP; 0 otherwise.</td>
</tr>
<tr>
<td>CODEPAGE</td>
<td>SMALLINT</td>
<td></td>
<td>Code page of the global variable.</td>
</tr>
<tr>
<td>COLLATIONSHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the collation for the variable.</td>
</tr>
<tr>
<td>COLLATIONNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the collation for the variable.</td>
</tr>
<tr>
<td>COLLATIONSHEMA_ORDERBY</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the collation for ORDER BY clauses in the variable.</td>
</tr>
<tr>
<td>COLLATIONNAME_ORDERBY</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the collation for ORDER BY clauses in the variable.</td>
</tr>
</tbody>
</table>
### Table 188. SYSCAT.VARIABLES Catalog View (continued)

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCOPE</td>
<td>CHAR (1)</td>
<td></td>
<td>Scope of the global variable.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• S = Session</td>
</tr>
<tr>
<td>DEFAULT</td>
<td>CLOB (64K)</td>
<td>Y</td>
<td>Expression used to calculate the initial value of the global variable when first referenced.</td>
</tr>
<tr>
<td>QUALIFIER</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>Value of the default schema at the time the variable was defined.</td>
</tr>
<tr>
<td>FUNC_PATH</td>
<td>CLOB (2K)</td>
<td>Y</td>
<td>SQL path in effect when the variable was defined.</td>
</tr>
<tr>
<td>REMARKS</td>
<td>VARCHAR (254)</td>
<td>Y</td>
<td>User-provided comments, or the null value.</td>
</tr>
<tr>
<td>READONLY</td>
<td>CHAR (1)</td>
<td></td>
<td>• C = Read-only because the global variable is defined with a CONSTANT clause</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = Not read-only</td>
</tr>
</tbody>
</table>
SYSCAT.VIEWS

Each row represents a view or materialized query table.

Table 189. SYSCAT.VIEWS Catalog View

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VIEWSCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the view or materialized query table.</td>
</tr>
<tr>
<td>VIEWNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the view or materialized query table.</td>
</tr>
<tr>
<td>OWNER</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Authorization ID of the owner of the view or materialized query table.</td>
</tr>
<tr>
<td>OWNERTYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>• S = The owner is the system</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• U = The owner is an individual user</td>
</tr>
<tr>
<td>SEQNO</td>
<td>SMALLINT</td>
<td></td>
<td>Always 1.</td>
</tr>
<tr>
<td>VIEWCHECK</td>
<td>CHAR (1)</td>
<td></td>
<td>Type of view checking.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• C = Cascaded check option</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• L = Local check option</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = No check option or is a materialized query table</td>
</tr>
<tr>
<td>READMEONLY</td>
<td>CHAR (1)</td>
<td></td>
<td>• N = View can be updated by users with appropriate authorization or is a materialized query table</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = View is read-only because of its definition</td>
</tr>
<tr>
<td>VALID</td>
<td>CHAR (1)</td>
<td></td>
<td>• N = View or materialized query table definition is invalid</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• X = View or materialized query table definition is inoperative and must be recreated</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = View or materialized query table definition is valid</td>
</tr>
<tr>
<td>QUALIFIER</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Value of the default schema at the time of object definition. Used to complete any unqualified references.</td>
</tr>
<tr>
<td>FUNC_PATH</td>
<td>CLOB (2K)</td>
<td></td>
<td>SQL path in effect when the view or materialized query table was defined.</td>
</tr>
<tr>
<td>TEXT</td>
<td>CLOB (2M)</td>
<td></td>
<td>Full text of the view or materialized query table CREATE statement, exactly as typed.</td>
</tr>
<tr>
<td>DEFINER(^1)</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Authorization ID of the owner of the view or materialized query table.</td>
</tr>
</tbody>
</table>

Note:
1. The DEFINER column is included for backwards compatibility. See OWNER.
### SYSCAT.WORKACTIONS

Each row represents a work action that is defined for a work action set.

#### Table 190. SYSCAT.WORKACTIONS Catalog View

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACTIONNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Name of the work action.</td>
</tr>
<tr>
<td>ACTIONID</td>
<td>INTEGER</td>
<td></td>
<td>Identifier for the work action.</td>
</tr>
<tr>
<td>ACTIONSETNAME</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>Name of the work action set.</td>
</tr>
<tr>
<td>ACTIONSETID</td>
<td>INTEGER</td>
<td></td>
<td>Identifier of the work action set to which this work action belongs. This column refers to the ACTIONSETID column in the SYSCAT.WORKACTIONSETS view.</td>
</tr>
<tr>
<td>WORKCLASSNAME</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>Name of the work class.</td>
</tr>
<tr>
<td>WORKCLASSID</td>
<td>INTEGER</td>
<td></td>
<td>Identifier of the work class. This column refers to the WORKCLASSID column in the SYSCAT.WORKCLASSES view.</td>
</tr>
<tr>
<td>CREATE_TIME</td>
<td>TIMESTAMP</td>
<td></td>
<td>Time at which the work action was created.</td>
</tr>
<tr>
<td>ALTER_TIME</td>
<td>TIMESTAMP</td>
<td></td>
<td>Time at which the work action was last altered.</td>
</tr>
<tr>
<td>ENABLED</td>
<td>CHAR (1)</td>
<td></td>
<td>- N = This work action is disabled.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Y = This work action is enabled.</td>
</tr>
</tbody>
</table>
### SYSCAT.WORKACTIONS

**Table 190. SYSCAT.WORKACTIONS Catalog View (continued)**

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACTIONTYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>The type of action performed on each DB2 activity that matches the attributes in the work class within scope.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- B = Collect basic aggregate activity data, specifiable only for work action sets that apply to service classes or workloads.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- C = Allow any DB2 activity under the associated work class to execute and increment the work class counter.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- D = Collect activity data with details at the database partition of the coordinator of the activity.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- E = Collect extended aggregate activity data, specifiable only for work action sets that apply to service classes or workloads.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- F = Collect activity data with details, section, and values at the database partition of the coordinator of the activity.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- G = Collect activity details and section at the database partition of the coordinator of the activity and collect activity data at all database partitions.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- H = Collect activity details, section, and values at the database partition of the coordinator of the activity and collect activity data at all database partitions.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- M = Map to a service subclass, specifiable only for work action sets that apply to service classes.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- P = Prevent the execution of any DB2 activity under the work class with which this work action is associated.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- S = Collect activity data with details and section at the database partition of the coordinator of the activity.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- T = The action represents a threshold, specifiable only for work action sets that are associated with a database or a workload.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- U = Map all activities with a nesting level of zero and all activities nested under these activities to a service subclass, specifiable only for work action sets that apply to service classes.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- V = Collect activity data with details and values at the coordinator partition.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- W = Collect activity data without details at the coordinator partition.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- X = Collect activity data with details at the coordinator partition and collect activity data at all database partitions.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Y = Collect activity data with details and values at the coordinator partition and collect activity data at all database partitions.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Z = Collect activity data without details at all database partitions.</td>
</tr>
</tbody>
</table>

<p>| REFOBJECTID   | INTEGER     | Y        | If ACTIONTYPE is 'M' (map) or 'N' (map nested), this value is set to the ID of the service subclass to which the DB2 activity is mapped. If ACTIONTYPE is 'T' (threshold), this value is set to the ID of the threshold to be used. For all other actions, this value is NULL. |</p>
<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>REFOBJECTTYPE</td>
<td>VARCHAR(30)</td>
<td></td>
<td>If the ACTIONTYPE is 'M' or 'N', this value is set to 'SERVICE CLASS'; if the ACTIONTYPE is 'T', this value is 'THRESHOLD'; the null value otherwise.</td>
</tr>
</tbody>
</table>
## SYSCAT.WORKACTIONSETs

Each row represents a work action set.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACTIONSETNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Name of the work action set.</td>
</tr>
<tr>
<td>ACTIONSETID</td>
<td>INTEGER</td>
<td></td>
<td>Identifier for the work action set.</td>
</tr>
<tr>
<td>WORKCLASSSETNAME</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>Name of the work class set.</td>
</tr>
<tr>
<td>WORKCLASSSETID</td>
<td>INTEGER</td>
<td></td>
<td>The identifier of the work class set that is to be mapped to the object specified by the OBJECTID. This column refers to WORKCLASSSETID in the SYSCAT.WORKCLASSSETs view.</td>
</tr>
<tr>
<td>CREATE_TIME</td>
<td>TIMESTAMP</td>
<td></td>
<td>Time at which the work action set was created.</td>
</tr>
<tr>
<td>ALTER_TIME</td>
<td>TIMESTAMP</td>
<td></td>
<td>Time at which the work action set was last altered.</td>
</tr>
<tr>
<td>ENABLED</td>
<td>CHAR (1)</td>
<td></td>
<td>- N = This work action set is disabled.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Y = This work action set is enabled.</td>
</tr>
<tr>
<td>OBJECTTYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>- b = Service superclass</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- w = Workload</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Blank = Database</td>
</tr>
<tr>
<td>OBJECTNAME</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>Name of the service class or workload.</td>
</tr>
<tr>
<td>OBJECTID</td>
<td>INTEGER</td>
<td></td>
<td>The identifier of the object to which the work class set (specified by the WORKCLASSSETID) is mapped. If the OBJECTTYPE is 'b', the OBJECTID is the ID of the service superclass. If the OBJECTTYPE is 'w', the OBJECTID is the ID of the workload. If the OBJECTTYPE is blank, the OBJECTID is -1.</td>
</tr>
<tr>
<td>REMARKS</td>
<td>VARCHAR (254)</td>
<td>Y</td>
<td>User-provided comments, or the null value.</td>
</tr>
</tbody>
</table>
**SYSCAT.WORKCLASSES**

Each row represents a work class defined for a work class set.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WORKCLASSNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Name of the work class.</td>
</tr>
<tr>
<td>WORKCLASSSETNAME</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>Name of the work class set.</td>
</tr>
<tr>
<td>WORKCLASSID</td>
<td>INTEGER</td>
<td></td>
<td>Identifier for the work class.</td>
</tr>
<tr>
<td>WORKCLASSSETID</td>
<td>INTEGER</td>
<td></td>
<td>Identifier for the work class set to which this work class belongs. This column refers to the WORKCLASSSETID column in the SYSCAT.WORKCLASSSETS view.</td>
</tr>
<tr>
<td>CREATE_TIME</td>
<td>TIMESTAMP</td>
<td></td>
<td>Time at which the work class was created.</td>
</tr>
<tr>
<td>ALTER_TIME</td>
<td>TIMESTAMP</td>
<td></td>
<td>Time at which the work class was last altered.</td>
</tr>
<tr>
<td>WORKTYPE</td>
<td>SMALLINT</td>
<td></td>
<td>The type of DB2 activity.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• 1 = ALL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• 2 = READ</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• 3 = WRITE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• 4 = CALL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• 5 = DML</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• 6 = DDL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• 7 = LOAD</td>
</tr>
<tr>
<td>RANGEUNITS</td>
<td>CHAR (1)</td>
<td></td>
<td>The units to use for the bottom and top range.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• C = Cardinality</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• T = Timerons</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Blank = Not applicable</td>
</tr>
<tr>
<td>FROMVALUE</td>
<td>DOUBLE</td>
<td>Y</td>
<td>The low value of the range in the units specified by the RANGEUNITS. Null value when RANGEUNITS is blank.</td>
</tr>
<tr>
<td>TOVALUE</td>
<td>DOUBLE</td>
<td>Y</td>
<td>The high value of the range in the units specified by the RANGEUNITS. Null value when RANGEUNITS is blank. -1 value is used to indicate no upper bound.</td>
</tr>
<tr>
<td>ROUTINESCHEMA</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>Schema name of the procedures that are called from the CALL statement. Null value when WORKTYPE is not 4 (CALL) or 1 (ALL).</td>
</tr>
<tr>
<td>INITIALSQLDATAPRIORITY</td>
<td>CHAR (1)</td>
<td></td>
<td>Reserved for future use.</td>
</tr>
<tr>
<td>EVALUATIONORDER</td>
<td>SMALLINT</td>
<td></td>
<td>Uniquely identifies the evaluation order used for choosing a work class within a work class set.</td>
</tr>
</tbody>
</table>
SYSCAT.WORKCLASSSETS

Each row represents a work class set.

Table 193. SYSCAT.WORKCLASSSETS Catalog View

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WORKCLASSSETNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Name of the work class set.</td>
</tr>
<tr>
<td>WORKCLASSSETID</td>
<td>INTEGER</td>
<td></td>
<td>Identifier for the work class set.</td>
</tr>
<tr>
<td>CREATE_TIME</td>
<td>TIMESTAMP</td>
<td></td>
<td>Time at which the work class set was created.</td>
</tr>
<tr>
<td>ALTER_TIME</td>
<td>TIMESTAMP</td>
<td></td>
<td>Time at which the work class set was last altered.</td>
</tr>
<tr>
<td>REMARKS</td>
<td>VARCHAR (254)</td>
<td>Y</td>
<td>User-provided comments, or the null value.</td>
</tr>
</tbody>
</table>
SYSCAT.WORKLOADAUTH

Each row represents a user, group, or role that has been granted USAGE privilege on a workload.

Table 194. SYSCAT.WORKLOADAUTH Catalog View

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WORKLOADID</td>
<td>INTEGER</td>
<td></td>
<td>Identifier for the workload.</td>
</tr>
<tr>
<td>WORKLOADNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Name of the workload.</td>
</tr>
<tr>
<td>GRANTOR</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Grantor of the privilege.</td>
</tr>
<tr>
<td>GRANTORTYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>• U = Grantee is an individual user</td>
</tr>
<tr>
<td>GRANTEE</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Holder of the privilege.</td>
</tr>
<tr>
<td>GRANTEEETYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>• G = Grantee is a group</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• R = Grantee is a role</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• U = Grantee is an individual user</td>
</tr>
<tr>
<td>USAGEAUTH</td>
<td>CHAR (1)</td>
<td></td>
<td>Indicates whether grantee holds USAGE privilege on the workload.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = Not held</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = Held</td>
</tr>
</tbody>
</table>

Appendix D. Catalog views 945
Each row represents a connection attribute in the definition of a workload.

Table 195. SYSCAT.WORKLOADCONNATTR Catalog View

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WORKLOADID</td>
<td>INTEGER</td>
<td></td>
<td>Identifier for the workload.</td>
</tr>
<tr>
<td>WORKLOADNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Name of the workload.</td>
</tr>
<tr>
<td>CONNATTRTYPE</td>
<td>VARCHAR (30)</td>
<td></td>
<td>Type of the connection attribute.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• 1 = APPLNAME</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• 2 = SYSTEM_USER</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• 3 = SESSION_USER</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• 4 = SESSION_USER GROUP</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• 5 = SESSION_USER ROLE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• 6 = CURRENT CLIENT_USERID</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• 7 = CURRENT CLIENT_APPLNAME</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• 8 = CURRENT CLIENT_WRKSTNNNAME</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• 9 = CURRENT CLIENT_ACCTNG</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• 10 = ADDRESS</td>
</tr>
<tr>
<td>CONNATTRVALUE</td>
<td>VARCHAR (1000)</td>
<td></td>
<td>Value of the connection attribute.</td>
</tr>
</tbody>
</table>
SYSCAT.WORKLOADS

Each row represents a workload.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WORKLOADID</td>
<td>INTEGER</td>
<td></td>
<td>Identifier for the workload.</td>
</tr>
<tr>
<td>WORKLOADNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Name of the workload.</td>
</tr>
<tr>
<td>EVALUATIONORDER</td>
<td>SMALLINT</td>
<td></td>
<td>Evaluation order used for choosing a workload.</td>
</tr>
<tr>
<td>CREATE_TIME</td>
<td>TIMESTAMP</td>
<td></td>
<td>Time at which the workload was created.</td>
</tr>
<tr>
<td>ALTER_TIME</td>
<td>TIMESTAMP</td>
<td></td>
<td>Time at which the workload was last altered.</td>
</tr>
<tr>
<td>ENABLED</td>
<td>CHAR (1)</td>
<td></td>
<td>• N = This workload is disabled.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = This workload is enabled.</td>
</tr>
<tr>
<td>ALLOWACCESS</td>
<td>CHAR (1)</td>
<td></td>
<td>• N = A UOW associated with this workload will be rejected.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = A unit of work (UOW) associated with this workload can access the database.</td>
</tr>
<tr>
<td>SERVICECLASSNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Name of the service subclass to which a unit of work (associated with this workload) is assigned.</td>
</tr>
<tr>
<td>PARENTSERVICECLASSNAME</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>Name of the service superclass to which a unit of work (associated with this workload) is assigned.</td>
</tr>
<tr>
<td>COLLECTAGGACTDATA</td>
<td>CHAR (1)</td>
<td></td>
<td>Specifies what aggregate activity data should be captured for the workload by the applicable event monitor.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• B = Collect base aggregate activity data</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• E = Collect extended aggregate activity data</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = None</td>
</tr>
<tr>
<td>COLLECTACTDATA</td>
<td>CHAR (1)</td>
<td></td>
<td>Specifies what activity data should be collected by the applicable event monitor.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• D = Activity data with details</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = None</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• S = Activity data with details and section environment</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• V = Activity data with details and values. Applies when the COLLECT column is set to ‘C’</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• W = Activity data without details</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• X = Activity data with details, section environment, and values</td>
</tr>
<tr>
<td>COLLECTACTPARTITION</td>
<td>CHAR (1)</td>
<td></td>
<td>Specifies where activity data is collected.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• C = Database partition of the coordinator of the activity</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• D = All database partitions</td>
</tr>
</tbody>
</table>
### SYSCAT.WORKLOADS

**Table 196. SYSCAT.WORKLOADS Catalog View (continued)**

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>COLLECTDEADLOCK</td>
<td>CHAR (1)</td>
<td></td>
<td>Specifies that deadlock events should be collect by the applicable event monitor.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• H = Collect deadlock data with past activities only</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = Do not collect deadlock data</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• V = Collect deadlock data with past activities and values</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• W = Collect deadlock data without past activities and values</td>
</tr>
<tr>
<td>COLLECTLOCKTIMEOUT</td>
<td>CHAR (1)</td>
<td></td>
<td>Specifies that lock timeout events should be collect by the applicable event monitor.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• H = Collect lock timeout data with past activities only</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = Do not collect lock timeout data</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• V = Collect lock timeout data with past activities and values</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• W = Collect lock timeout data without past activities and values</td>
</tr>
<tr>
<td>COLLECTLOCKWAIT</td>
<td>CHAR (1)</td>
<td></td>
<td>Specifies that lock wait events should be collect by the applicable event monitor.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• H = Collect lock wait data with past activities only</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = Do not collect lock wait data</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• V = Collect lock wait data with past activities and values</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• W = Collect lock wait data without past activities and values</td>
</tr>
<tr>
<td>LOCKWAITVALUE</td>
<td>INTEGER</td>
<td></td>
<td>Specifies the time in milliseconds a lock should wait before a lock event is collected by the applicable event monitor; 0 when COLLECTLOCKWAIT = 'N'</td>
</tr>
<tr>
<td>COLLECTACTMETRICS</td>
<td>CHAR (1)</td>
<td></td>
<td>Specifies the monitoring level for activities submitted by an occurrence of the workload.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• B = Collect base activity metrics</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• E = Collect extended activity metrics</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = None</td>
</tr>
<tr>
<td>COLLECTUOWDATA</td>
<td>CHAR (1)</td>
<td></td>
<td>Specifies what unit of work data should be collected by the applicable event monitor.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• B = Collect base unit of work data</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = None</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• P = Collect base unit of work data and the package list</td>
</tr>
<tr>
<td>EXTERNALNAME</td>
<td>VARCHAR (128)</td>
<td>Y</td>
<td>Reserved for future use.</td>
</tr>
<tr>
<td>REMARKS</td>
<td>VARCHAR (254)</td>
<td>Y</td>
<td>User-provided comments, or the null value.</td>
</tr>
</tbody>
</table>
Each row represents a wrapper-specific option.

### Table 197. SYSCAT.WRAPOPTIONS Catalog View

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WRAPNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Name of the wrapper.</td>
</tr>
<tr>
<td>OPTION</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Name of the wrapper option.</td>
</tr>
<tr>
<td>SETTING</td>
<td>VARCHAR (2048)</td>
<td></td>
<td>Value of the wrapper option.</td>
</tr>
</tbody>
</table>
Each row represents a registered wrapper.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WRAPNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Name of the wrapper.</td>
</tr>
<tr>
<td>WRAPTYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>Type of wrapper.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = Non-relational</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• R = Relational</td>
</tr>
<tr>
<td>WRAPVERSION</td>
<td>INTEGER</td>
<td></td>
<td>Version of the wrapper.</td>
</tr>
<tr>
<td>LIBRARY</td>
<td>VARCHAR (255)</td>
<td></td>
<td>Name of the file that contains the code used to communicate with the data</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>sources that are associated with this wrapper.</td>
</tr>
<tr>
<td>REMARKS</td>
<td>VARCHAR (254)</td>
<td>Y</td>
<td>User-provided comments, or the null value.</td>
</tr>
</tbody>
</table>
SYSCAT.XDBMAPGRAPHS

Each row represents a schema graph for an XDB map (XSR object).

**Table 199. SYSCAT.XDBMAPGRAPHS Catalog View**

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OBJECTID</td>
<td>BIGINT</td>
<td></td>
<td>Unique generated identifier for an XSR object.</td>
</tr>
<tr>
<td>OBJECTSCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the XSR object.</td>
</tr>
<tr>
<td>OBJECTNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the XSR object.</td>
</tr>
<tr>
<td>SCHEMAGRAPHID</td>
<td>INTEGER</td>
<td></td>
<td>Schema graph identifier, which is unique within an XDB map identifier.</td>
</tr>
<tr>
<td>NAMESPACE</td>
<td>VARCHAR (1001)</td>
<td>Y</td>
<td>String identifier for the namespace URI of the root element.</td>
</tr>
<tr>
<td>ROOTELEMENT</td>
<td>VARCHAR (1001)</td>
<td>Y</td>
<td>String identifier for the element name of the root element.</td>
</tr>
</tbody>
</table>
SYSCAT.XDBMAPSHREDTREES

Each row represents one shred tree for a given schema graph identifier.

Table 200. SYSCAT.XDBMAPSHREDTREES Catalog View

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OBJECTID</td>
<td>BIGINT</td>
<td></td>
<td>Unique generated identifier for an XSR object.</td>
</tr>
<tr>
<td>OBJECTSCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the XSR object.</td>
</tr>
<tr>
<td>OBJECTNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the XSR object.</td>
</tr>
<tr>
<td>SCHEMAGRAPHID</td>
<td>INTEGER</td>
<td></td>
<td>Schema graph identifier, which is unique within an XDB map identifier.</td>
</tr>
<tr>
<td>SHREDTREEID</td>
<td>INTEGER</td>
<td></td>
<td>Shred tree identifier, which is unique within an XDB map identifier.</td>
</tr>
<tr>
<td>MAPPINGDESCRIPTION</td>
<td>CLOB (1M)</td>
<td>Y</td>
<td>Diagnostic mapping information.</td>
</tr>
</tbody>
</table>
Each row represents a single string and its unique string ID, used to condense structural XML data. The string is provided in both UTF-8 encoding and database codepage encoding.

Table 201. SYSCAT.XMLSTRINGS Catalog View

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STRINGID</td>
<td>INTEGER</td>
<td></td>
<td>Unique string ID.</td>
</tr>
<tr>
<td>STRING</td>
<td>VARCHAR(1001)</td>
<td></td>
<td>The string represented in the database codepage.</td>
</tr>
<tr>
<td>STRING_UTF8</td>
<td>VARCHAR(1001)</td>
<td></td>
<td>The string in UTF-8 encoding (as stored in the catalog table).</td>
</tr>
</tbody>
</table>
SYSCAT.XSROBJECTAUTH

Each row represents a user, group, or role that has been granted the USAGE privilege on a particular XSR object.

Table 202. SYSCAT.XSROBJECTAUTH Catalog View

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRANTOR</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Grantor of the privilege.</td>
</tr>
<tr>
<td>GRANTORTYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>• S = Grantor is the system</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• U = Grantor is an individual user</td>
</tr>
<tr>
<td>GRANTEE</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Holder of the privilege.</td>
</tr>
<tr>
<td>GRANTEETYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>• G = Grantee is a group</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• R = Grantee is a role</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• U = Grantee is an individual user</td>
</tr>
<tr>
<td>OBJECTID</td>
<td>BIGINT</td>
<td></td>
<td>Identifier for the XSR object.</td>
</tr>
<tr>
<td>USAGEAUTH</td>
<td>CHAR (1)</td>
<td></td>
<td>Privilege to use the XSR object and its components.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = Not held</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = Held</td>
</tr>
</tbody>
</table>
Each row represents an XSR object component.

**Table 203. SYSCAT.XSROBJECTCOMPONENTS Catalog View**

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OBJECTID</td>
<td>BIGINT</td>
<td></td>
<td>Unique generated identifier for an XSR object.</td>
</tr>
<tr>
<td>OBJECTSCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the XSR object.</td>
</tr>
<tr>
<td>OBJECTNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the XSR object.</td>
</tr>
<tr>
<td>COMPONENTID</td>
<td>BIGINT</td>
<td></td>
<td>Unique generated identifier for an XSR object component.</td>
</tr>
<tr>
<td>TARGETNAMESPACE</td>
<td>VARCHAR (1001)</td>
<td>Y</td>
<td>String identifier for the target namespace.</td>
</tr>
<tr>
<td>SCHEMALOCATION</td>
<td>VARCHAR (1001)</td>
<td>Y</td>
<td>String identifier for the schema location.</td>
</tr>
<tr>
<td>COMPONENT</td>
<td>BLOB (30M)</td>
<td>Y</td>
<td>External representation of the component.</td>
</tr>
<tr>
<td>CREATE_TIME</td>
<td>TIMESTAMP</td>
<td></td>
<td>Time at which the XSR object component was registered.</td>
</tr>
<tr>
<td>STATUS</td>
<td>CHAR (1)</td>
<td></td>
<td>Registration status.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• C = Complete</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• I = Incomplete</td>
</tr>
</tbody>
</table>
Each row represents a dependency of an XSR object on some other object. The XSR object depends on the object of type BTYPE of name BNAME, so a change to the object affects the XSR object.

Table 204. SYSCAT.XSROBJECTDEP Catalog View

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OBJECTID</td>
<td>BIGINT</td>
<td></td>
<td>Unique generated identifier for an XSR object.</td>
</tr>
<tr>
<td>OBJECTSCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the XSR object.</td>
</tr>
<tr>
<td>OBJECTNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the XSR object.</td>
</tr>
<tr>
<td>BTYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>Type of object on which there is a dependency. Possible values are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• A = Table alias</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• B = Trigger</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• F = Routine</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• G = Global temporary table</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• H = Hierarchy table</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• K = Package</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• L = Detached table</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = Nickname</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• O = Privilege dependency on all subtables or subviews in a table or view hierarchy</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Q = Sequence</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• R = User-defined data type</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• S = Materialized query table</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• T = Table (not typed)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• U = Typed table</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• V = View (not typed)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• W = Typed view</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• X = Index extension</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Z = XSR object</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• q = Sequence alias</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• u = Module alias</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• v = Global variable</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• * = Anchored to the row of a base table</td>
</tr>
<tr>
<td>BSCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the object on which there is a dependency.</td>
</tr>
<tr>
<td>BMODULENAME</td>
<td>VARCHAR(128)</td>
<td>Y</td>
<td>Unqualified name of the module to which the object on which a dependency belongs. The null value if not a module object.</td>
</tr>
<tr>
<td>BNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the object on which there is a dependency. For routines (BTYPE = 'F'), this is the specific name.</td>
</tr>
<tr>
<td>BMODULEID</td>
<td>INTEGER</td>
<td>Y</td>
<td>Identifier for the module of the object on which there is a dependency.</td>
</tr>
<tr>
<td>Column Name</td>
<td>Data Type</td>
<td>Nullable</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>------------</td>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td>TABAUTH</td>
<td>SMALLINT</td>
<td>Y</td>
<td>If BTYPE = ‘O’, ‘S’, ‘T’, ‘U’, ‘V’, ‘W’, or ‘v’, encodes the privileges on the table or view that are required by a dependent trigger; null value otherwise.</td>
</tr>
</tbody>
</table>
Each row represents an XML schema repository object.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OBJECTID</td>
<td>BIGINT</td>
<td></td>
<td>Unique generated identifier for an XML schema object.</td>
</tr>
<tr>
<td>OBJECTSCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the XML schema object.</td>
</tr>
<tr>
<td>OBJECTNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the XML schema object.</td>
</tr>
<tr>
<td>GRAMMAR</td>
<td>BLOB (127M)</td>
<td>Y</td>
<td>Binary representation of the grammar for the XML schema object.</td>
</tr>
<tr>
<td>PROPERTIES</td>
<td>BLOB (4190000)</td>
<td>Y</td>
<td>Properties document for the XML schema object.</td>
</tr>
</tbody>
</table>
SYSCAT.XSROBJECTHIERARCHIES

Each row represents the hierarchical relationship between an XSR object and its components.

Table 206. SYSCAT.XSROBJECTHIERARCHIES Catalog View

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OBJECTID</td>
<td>BIGINT</td>
<td></td>
<td>Identifier for an XSR object.</td>
</tr>
<tr>
<td>COMPONENTID</td>
<td>BIGINT</td>
<td></td>
<td>Identifier for an XSR component.</td>
</tr>
<tr>
<td>HTYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>Hierarchy type.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>D = Document</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>N = Top-level namespace</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>P = Primary document</td>
</tr>
<tr>
<td>TARGETNAMESPACE</td>
<td>VARCHAR (1001)</td>
<td>Y</td>
<td>String identifier for the component’s target namespace.</td>
</tr>
<tr>
<td>SCHEMALLOCATION</td>
<td>VARCHAR (1001)</td>
<td>Y</td>
<td>String identifier for the component’s schema location.</td>
</tr>
</tbody>
</table>
Each row represents an XML schema repository object.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OBJECTID</td>
<td>BIGINT</td>
<td></td>
<td>Unique generated identifier for an XSR object.</td>
</tr>
<tr>
<td>OBJECTSCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Schema name of the XSR object.</td>
</tr>
<tr>
<td>OBJECTNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Unqualified name of the XSR object.</td>
</tr>
<tr>
<td>TARGETNAMESPACE</td>
<td>VARCHAR (1001)</td>
<td>Y</td>
<td>String identifier for the target namespace, or public identifier.</td>
</tr>
<tr>
<td>SCHEMALOCATION</td>
<td>VARCHAR (1001)</td>
<td>Y</td>
<td>String identifier for the schema location, or system identifier.</td>
</tr>
<tr>
<td>OBJECTINFO</td>
<td>XML</td>
<td>Y</td>
<td>Metadata document.</td>
</tr>
<tr>
<td>OBJECTTYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>XSR object type.</td>
</tr>
<tr>
<td>OWNER</td>
<td>VARCHAR (128)</td>
<td></td>
<td>Authorization ID of the owner of the XSR object.</td>
</tr>
<tr>
<td>OWNERTYPE</td>
<td>CHAR (1)</td>
<td></td>
<td>• S = The owner is the system</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• U = The owner is an individual user</td>
</tr>
<tr>
<td>CREATE_TIME</td>
<td>TIMESTAMP</td>
<td></td>
<td>Time at which the object was registered.</td>
</tr>
<tr>
<td>ALTER_TIME</td>
<td>TIMESTAMP</td>
<td></td>
<td>Time at which the object was last updated (replaced).</td>
</tr>
<tr>
<td>STATUS</td>
<td>CHAR (1)</td>
<td></td>
<td>Registration status.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• C = Complete</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• I = Incomplete</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• R = Replace</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• T = Temporary</td>
</tr>
<tr>
<td>DECOMPOSITION</td>
<td>CHAR (1)</td>
<td></td>
<td>Indicates whether or not decomposition (shredding) is enabled on this XSR</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>object.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• N = Not enabled</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• X = Inoperative</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Y = Enabled</td>
</tr>
<tr>
<td>REMARKS</td>
<td>VARCHAR (254)</td>
<td>Y</td>
<td>User-provided comments, or the null value.</td>
</tr>
</tbody>
</table>
SYSIBM.SYSDUMMY1

Contains one row. This view is available for applications that require compatibility with DB2 for z/OS.

**Table 208. SYSIBM.SYSDUMMY1 Catalog View**

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBMREQD</td>
<td>CHAR(1)</td>
<td>'Y'</td>
<td></td>
</tr>
</tbody>
</table>
SYSSTAT.COLDIST

Each row represents the \( n \)th most frequent value of some column, or the \( n \)th quantile (cumulative distribution) value of the column. Applies to columns of real tables only (not views). No statistics are recorded for inherited columns of typed tables.

Table 209. SYSSTAT.COLDIST Catalog View

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Updatable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TABSCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td></td>
<td>Schema name of the table to which the statistics apply.</td>
</tr>
<tr>
<td>TABNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td></td>
<td>Unqualified name of the table to which the statistics apply.</td>
</tr>
<tr>
<td>COLNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td></td>
<td>Name of the column to which the statistics apply.</td>
</tr>
<tr>
<td>TYPE</td>
<td>CHAR (1)</td>
<td></td>
<td></td>
<td>• ( F ) = Frequency value \n• ( Q ) = Quantile value</td>
</tr>
<tr>
<td>SEQNO</td>
<td>SMALLINT</td>
<td></td>
<td></td>
<td>If TYPE = 'F', ( n ) in this column identifies the ( n )th most frequent value. If TYPE = 'Q', ( n ) in this column identifies the ( n )th quantile value.</td>
</tr>
<tr>
<td>COLVALUE1</td>
<td>VARCHAR (254)</td>
<td>Y</td>
<td>Y</td>
<td>Data value as a character literal or a null value.</td>
</tr>
<tr>
<td>VALCOUNT</td>
<td>BIGINT</td>
<td>Y</td>
<td></td>
<td>If TYPE = 'F', VALCOUNT is the number of occurrences of COLVALUE in the column. If TYPE = 'Q', VALCOUNT is the number of rows whose value is less than or equal to COLVALUE.</td>
</tr>
<tr>
<td>DISTCOUNT2</td>
<td>BIGINT</td>
<td>Y</td>
<td>Y</td>
<td>If TYPE = 'Q', this column records the number of distinct values that are less than or equal to COLVALUE (the null value if unavailable).</td>
</tr>
</tbody>
</table>

Note:

1. In the catalog view, the value of COLVALUE is always shown in the database code page and can contain substitution characters. However, the statistics are gathered internally in the code page of the column’s table, and will therefore use actual column values when applied during query optimization.

2. DISTCOUNT is collected only for columns that are the first key column in an index.
SYSSTAT.COLGROUPDIST

Each row represents the value of the column in a column group that makes up the $n$th most frequent value of the column group or the $n$th quantile value of the column group.

Table 210. SYSSTAT.COLGROUPDIST Catalog View

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Updatable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>COLGROUPID</td>
<td>INTEGER</td>
<td></td>
<td></td>
<td>Identifier for the column group.</td>
</tr>
<tr>
<td>TYPE</td>
<td>CHAR (1)</td>
<td></td>
<td></td>
<td>• F = Frequency value</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Q = Quantile value</td>
</tr>
<tr>
<td>ORDINAL</td>
<td>SMALLINT</td>
<td></td>
<td></td>
<td>Ordinal number of the column in the column group.</td>
</tr>
<tr>
<td>SEQNO</td>
<td>SMALLINT</td>
<td></td>
<td></td>
<td>If TYPE = 'F', $n$ in this column identifies the $n$th most frequent value.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If TYPE = 'Q', $n$ in this column identifies the $n$th quantile value.</td>
</tr>
<tr>
<td>COLVALUE</td>
<td>VARCHAR (254)</td>
<td>Y</td>
<td></td>
<td>Data value as a character literal or a null value.</td>
</tr>
</tbody>
</table>
SYSSTAT.COLGROUPDISTCOUNTS

Each row represents the distribution statistics that apply to the \( n \)th most frequent value of a column group or the \( n \)th quantile of a column group.

Table 211. SYSSTAT.COLGROUPDISTCOUNTS Catalog View

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Updatable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>COLGROUPID</td>
<td>INTEGER</td>
<td></td>
<td></td>
<td>Identifier for the column group.</td>
</tr>
<tr>
<td>TYPE</td>
<td>CHAR (1)</td>
<td></td>
<td></td>
<td>• ( F ) = Frequency value</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• ( Q ) = Quantile value</td>
</tr>
<tr>
<td>SEQNO</td>
<td>SMALLINT</td>
<td></td>
<td></td>
<td>Sequence number ( n ) representing the ( n )th TYPE value.</td>
</tr>
<tr>
<td>VALCOUNT</td>
<td>BIGINT</td>
<td>Y</td>
<td></td>
<td>If TYPE = ( F ), VALCOUNT is the number of occurrences of COLVALUE for the column group with this SEQNO. If TYPE = ( Q ), VALCOUNT is the number of rows whose value is less than or equal to COLVALUE for the column group with this SEQNO.</td>
</tr>
<tr>
<td>DISTCOUNT</td>
<td>BIGINT</td>
<td>Y</td>
<td></td>
<td>If TYPE = ( Q ), this column records the number of distinct values that are less than or equal to COLVALUE for the column group with this SEQNO (the null value if unavailable).</td>
</tr>
</tbody>
</table>
Each row represents a column group and statistics that apply to the entire column group.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Updatable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>COLGROUPSCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td></td>
<td>Schema name of the column group.</td>
</tr>
<tr>
<td>COLGROUPNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td></td>
<td>Unqualified name of the column group.</td>
</tr>
<tr>
<td>COLGROUPID</td>
<td>INTEGER</td>
<td></td>
<td></td>
<td>Identifier for the column group.</td>
</tr>
<tr>
<td>COLGROUPCARD</td>
<td>BIGINT</td>
<td>Y</td>
<td></td>
<td>Cardinality of the column group.</td>
</tr>
<tr>
<td>NUMFREQ_VALUES</td>
<td>SMALLINT</td>
<td></td>
<td></td>
<td>Number of frequent values collected for the column group.</td>
</tr>
<tr>
<td>NUMQUANTILES</td>
<td>SMALLINT</td>
<td></td>
<td></td>
<td>Number of quantiles collected for the column group.</td>
</tr>
</tbody>
</table>
Each row represents a column defined for a table, view, or nickname.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Updatable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TABSCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td></td>
<td>Schema name of the table, view, or nickname that contains the column.</td>
</tr>
<tr>
<td>TABNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td></td>
<td>Unqualified name of the table, view, or nickname that contains the column.</td>
</tr>
<tr>
<td>COLNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td></td>
<td>Name of the column.</td>
</tr>
<tr>
<td>COLCARD</td>
<td>BIGINT</td>
<td></td>
<td>Y</td>
<td>Number of distinct values in the column; -1 if statistics are not collected; -2 for inherited columns and columns of hierarchy tables.</td>
</tr>
<tr>
<td>HIGH2KEYi²</td>
<td>VARCHAR (254)</td>
<td>Y</td>
<td>Y</td>
<td>Second-highest data value. Representation of numeric data changed to character literals. Empty if statistics are not collected. Empty for inherited columns and columns of hierarchy tables.</td>
</tr>
<tr>
<td>LOW2KEYi²</td>
<td>VARCHAR (254)</td>
<td>Y</td>
<td>Y</td>
<td>Second-lowest data value. Representation of numeric data changed to character literals. Empty if statistics are not collected. Empty for inherited columns and columns of hierarchy tables.</td>
</tr>
<tr>
<td>AVGCOLLEN</td>
<td>INTEGER</td>
<td></td>
<td>Y</td>
<td>Average space in bytes when the column is stored in database memory or a temporary table. For LOB data types that are not inlined, LONG data types, and XML documents, the value used to calculate the average column length is the length of the data descriptor. An extra byte is required if the column is nullable; -1 if statistics have not been collected; -2 for inherited columns and columns of hierarchy tables. Note: The average space required to store the column on disk may be different than the value represented by this statistic.</td>
</tr>
<tr>
<td>NUMNULLS</td>
<td>BIGINT</td>
<td></td>
<td>Y</td>
<td>Number of null values in the column; -1 if statistics are not collected.</td>
</tr>
<tr>
<td>PCTINLINED</td>
<td>SMALLINT</td>
<td></td>
<td></td>
<td>Percentage of inlined XML documents or LOB data. -1 if statistics have not been collected.</td>
</tr>
<tr>
<td>SUB_COUNT</td>
<td>SMALLINT</td>
<td></td>
<td>Y</td>
<td>Average number of sub-elements in the column. Applicable to character string columns only.</td>
</tr>
<tr>
<td>SUB_DELIM_LENGTH</td>
<td>SMALLINT</td>
<td></td>
<td>Y</td>
<td>Average length of the delimiters that separate each sub-element in the column. Applicable to character string columns only.</td>
</tr>
</tbody>
</table>
### Table 213. SYSSTAT.COLUMNS Catalog View (continued)

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Updatable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVGCOLLENCAP</td>
<td>INTEGER</td>
<td></td>
<td>Y</td>
<td>Average number of characters (based on the collation in effect for the column) required for the column; -1 if the data type of the column is long, LOB, or XML or if statistics have not been collected; -2 for inherited columns and columns of hierarchy tables.</td>
</tr>
</tbody>
</table>

**Note:**

1. In the catalog view, the values of HIGH2KEY and LOW2KEY are always shown in the database code page and can contain substitution characters. However, the statistics are gathered internally in the code page of the column’s table, and will therefore use actual column values when applied during query optimization.
SYSSTAT.INDEXES

Each row represents an index. Indexes on typed tables are represented by two rows: one for the "logical index" on the typed table, and one for the "H-index" on the hierarchy table.

Table 214. SYSSTAT.INDEXES Catalog View

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Updatable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INDSCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td></td>
<td>Schema name of the index.</td>
</tr>
<tr>
<td>INDXNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td></td>
<td>Unqualified name of the index.</td>
</tr>
<tr>
<td>TABSCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td></td>
<td>Schema name of the table or nickname on which the index is defined.</td>
</tr>
<tr>
<td>TABNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td></td>
<td>Unqualified name of the table or nickname on which the index is defined.</td>
</tr>
<tr>
<td>COLNAMES</td>
<td>VARCHAR (640)</td>
<td></td>
<td></td>
<td>This column is no longer used and will be removed in the next release.</td>
</tr>
<tr>
<td>NLEAF</td>
<td>BIGINT</td>
<td></td>
<td>Y</td>
<td>Number of leaf pages; -1 if statistics are not collected.</td>
</tr>
<tr>
<td>NLEVELS</td>
<td>SMALLINT</td>
<td></td>
<td>Y</td>
<td>Number of index levels; -1 if statistics are not collected.</td>
</tr>
<tr>
<td>FIRSTKEYCARD</td>
<td>BIGINT</td>
<td></td>
<td>Y</td>
<td>Number of distinct first-key values; -1 if statistics are not collected.</td>
</tr>
<tr>
<td>FIRST2KEYCARD</td>
<td>BIGINT</td>
<td></td>
<td>Y</td>
<td>Number of distinct keys using the first two columns of the index; -1 if</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>statistics are not collected, or if not applicable.</td>
</tr>
<tr>
<td>FIRST3KEYCARD</td>
<td>BIGINT</td>
<td></td>
<td>Y</td>
<td>Number of distinct keys using the first three columns of the index; -1 if</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>statistics are not collected, or if not applicable.</td>
</tr>
<tr>
<td>FIRST4KEYCARD</td>
<td>BIGINT</td>
<td></td>
<td>Y</td>
<td>Number of distinct keys using the first four columns of the index; -1 if</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>statistics are not collected, or if not applicable.</td>
</tr>
<tr>
<td>FULLKEYCARD</td>
<td>BIGINT</td>
<td></td>
<td>Y</td>
<td>Number of distinct full-key values; -1 if statistics are not collected.</td>
</tr>
<tr>
<td>CLUSTERRATIO</td>
<td>SMALLINT</td>
<td></td>
<td>Y</td>
<td>Degree of data clustering with the index; -1 if statistics are not collected</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>or if detailed index statistics are collected (in which case, CLUSTERFACTO</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>R will be used instead).</td>
</tr>
<tr>
<td>CLUSTERFACTOR</td>
<td>DOUBLE</td>
<td></td>
<td>Y</td>
<td>Finer measurement of the degree of clustering; -1 if statistics are not</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>collected or if the index is defined on a nickname.</td>
</tr>
<tr>
<td>SEQUENTIAL_PAGES</td>
<td>BIGINT</td>
<td></td>
<td>Y</td>
<td>Number of leaf pages located on disk in index key order with few or no</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>large gaps between them; -1 if statistics are not collected.</td>
</tr>
<tr>
<td>Column Name</td>
<td>Data Type</td>
<td>Nullable</td>
<td>Updatable</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-------------------</td>
<td>----------</td>
<td>-----------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>DENSITY</td>
<td>INTEGER</td>
<td>Y</td>
<td></td>
<td>Ratio of SEQUENTIAL_PAGES to number of pages in the range of pages occupied by the index, expressed as a percent (integer between 0 and 100); -1 if statistics are not collected.</td>
</tr>
<tr>
<td>PAGE_FETCH_PAIRS&lt;sup&gt;4&lt;/sup&gt;</td>
<td>VARCHAR (520)</td>
<td>Y</td>
<td></td>
<td>A list of pairs of integers, represented in character form. Each pair represents the number of pages in a hypothetical buffer, and the number of page fetches required to scan the table with this index using that hypothetical buffer. Zero-length string if no data is available.</td>
</tr>
<tr>
<td>NUMRIDS&lt;sup&gt;4&lt;/sup&gt;</td>
<td>BIGINT</td>
<td>Y</td>
<td></td>
<td>Total number of row identifiers (RIDs) or block identifiers (BIDs) in the index; -1 if not known.</td>
</tr>
<tr>
<td>NUMRIDS_DELETED&lt;sup&gt;4&lt;/sup&gt;</td>
<td>BIGINT</td>
<td>Y</td>
<td></td>
<td>Total number of row identifiers (or block identifiers) in the index that are marked deleted, excluding those identifiers on leaf pages on which all the identifiers are marked deleted.</td>
</tr>
<tr>
<td>NUM_EMPTY_LEAFS</td>
<td>BIGINT</td>
<td>Y</td>
<td></td>
<td>Total number of index leaf pages that have all of their row identifiers (or block identifiers) marked deleted.</td>
</tr>
<tr>
<td>AVERAGE_RANDOM_FETCH_PAGES&lt;sup&gt;1,2,4&lt;/sup&gt;</td>
<td>DOUBLE</td>
<td>Y</td>
<td></td>
<td>Average number of random table pages between sequential page accesses when fetching using the index; -1 if not known.</td>
</tr>
<tr>
<td>AVERAGE_RANDOM_PAGES&lt;sup&gt;5&lt;/sup&gt;</td>
<td>DOUBLE</td>
<td>Y</td>
<td></td>
<td>Average number of random table pages between sequential page accesses; -1 if not known.</td>
</tr>
<tr>
<td>AVERAGE_SEQUENCE_GAP&lt;sup&gt;2&lt;/sup&gt;</td>
<td>DOUBLE</td>
<td>Y</td>
<td></td>
<td>Gap between index page sequences. Detected through a scan of index leaf pages, each gap represents the average number of index pages that must be randomly fetched between sequences of index pages; -1 if not known.</td>
</tr>
<tr>
<td>AVERAGE_SEQUENCE_FETCH_GAP&lt;sup&gt;1,2,4&lt;/sup&gt;</td>
<td>DOUBLE</td>
<td>Y</td>
<td></td>
<td>Gap between table page sequences when fetching using the index. Detected through a scan of index leaf pages, each gap represents the average number of table pages that must be randomly fetched between sequences of table pages; -1 if not known.</td>
</tr>
<tr>
<td>AVERAGE_SEQUENCE_PAGES&lt;sup&gt;5&lt;/sup&gt;</td>
<td>DOUBLE</td>
<td>Y</td>
<td></td>
<td>Average number of index pages that are accessible in sequence (that is, the number of index pages that the prefachers would detect as being in sequence); -1 if not known.</td>
</tr>
</tbody>
</table>
### SYSSTAT.INDEXES

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Updatable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVERAGE_SEQUENCE_FETCH_PAGES</td>
<td>DOUBLE</td>
<td>Y</td>
<td></td>
<td>Average number of table pages that are accessible in sequence (that is, the number of table pages that the prefetchers would detect as being in sequence) when fetching using the index; -1 if not known.</td>
</tr>
<tr>
<td>AVGPARTITION_CLUSTERRATIO</td>
<td>SMALLINT</td>
<td>Y</td>
<td></td>
<td>Degree of data clustering within a single data partition. -1 if the table is not partitioned, if statistics are not collected, or if detailed statistics are collected (in which case AVGPARTITION_CLUSTERFACTOR will be used instead).</td>
</tr>
<tr>
<td>AVGPARTITION_CLUSTERFACTOR</td>
<td>DOUBLE</td>
<td>Y</td>
<td></td>
<td>Finer measurement of the degree of clustering within a single data partition. -1 if the table is not partitioned, if statistics are not collected, or if the index is defined on a nickname.</td>
</tr>
<tr>
<td>AVGPARTITION_PAGE_FETCH_PAIRS</td>
<td>VARCHAR (520)</td>
<td>Y</td>
<td></td>
<td>A list of paired integers in character form. Each pair represents a potential buffer pool size and the corresponding page fetches required to access a single data partition from the table. Zero-length string if no data is available, or if the table is not partitioned.</td>
</tr>
<tr>
<td>DATAPARTITION_CLUSTERFACTOR</td>
<td>DOUBLE</td>
<td>Y</td>
<td></td>
<td>A statistic measuring the &quot;clustering&quot; of the index keys with regard to data partitions. It is a number between 0 and 1, with 1 representing perfect clustering and 0 representing no clustering.</td>
</tr>
<tr>
<td>INDCARD</td>
<td>BIGINT</td>
<td>Y</td>
<td></td>
<td>Cardinality of the index. This might be different from the cardinality of the table for indexes that do not have a one-to-one relationship between the table rows and the index entries.</td>
</tr>
<tr>
<td>PCTPAGESAVED</td>
<td>SMALLINT</td>
<td></td>
<td></td>
<td>Approximate percentage of pages saved in the index as a result of index compression. -1 if statistics are not collected.</td>
</tr>
<tr>
<td>AVGLEAFKEYSIZE</td>
<td>INTEGER</td>
<td>Y</td>
<td></td>
<td>Average index key size for keys on leaf pages in the index.</td>
</tr>
<tr>
<td>AVGNLEAFKEYSIZE</td>
<td>INTEGER</td>
<td>Y</td>
<td></td>
<td>Average index key size for keys on non-leaf pages in the index.</td>
</tr>
</tbody>
</table>
Table 214. SYSSTAT.INDEXES Catalog View (continued)

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Updatable</th>
<th>Description</th>
</tr>
</thead>
</table>

Note:
1. When using DMS table spaces, this statistic cannot be computed.
2. Prefetch statistics are not gathered during a LOAD...STATISTICS YES, or a CREATE INDEX...COLLECT STATISTICS operation, or when the database configuration parameter `seqdetect` is turned off.
3. `AVGPARTITION_CLUSTERRATIO`, `AVGPARTITION_CLUSTERFACTOR`, and `AVGPARTITION_PAGE_FETCH_PAIRS` measure the degree of clustering within a single data partition (local clustering). `CLUSTERRATIO`, `CLUSTERFACTOR`, and `PAGE_FETCH_PAIRS` measure the degree of clustering in the entire table (global clustering). Global clustering and local clustering values can diverge significantly if the table partitioning key is not a prefix of the index key, or when the table partitioning key and the index key are logically independent of each other.
4. This statistic cannot be updated if the index type is ‘XPTH’ (an XML path index).
5. Because logical indexes on an XML column do not have statistics, the SYSSTAT.INDEXES catalog view excludes rows whose index type is ‘XVIL’.
SYSSTAT.ROUTINES

Each row represents a user-defined routine (scalar function, table function, sourced function, method, or procedure). Does not include built-in functions.

Table 215. SYSSTAT.ROUTINES Catalog View

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Updatable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROUTINESCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td></td>
<td>Schema name of the routine if ROUTINEMODULENAME is null; otherwise schema name of the module to which the routine belongs.</td>
</tr>
<tr>
<td>ROUTINEMODULENAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td></td>
<td>Unqualified name of the module to which the routine belongs. The null value if not a module routine.</td>
</tr>
<tr>
<td>ROUTINENAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td></td>
<td>Unqualified name of the routine.</td>
</tr>
<tr>
<td>ROUTINETYPE</td>
<td>CHAR (1)</td>
<td></td>
<td></td>
<td>Type of routine.</td>
</tr>
<tr>
<td>SPECIFICNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td></td>
<td>Name of the routine instance (might be system-generated).</td>
</tr>
<tr>
<td>IOS_PER_INVOC</td>
<td>DOUBLE</td>
<td>Y</td>
<td></td>
<td>Estimated number of inputs/outputs (I/Os) per invocation; 0 is the default; -1 if not known.</td>
</tr>
<tr>
<td>INSTS_PER_INVOC</td>
<td>DOUBLE</td>
<td>Y</td>
<td></td>
<td>Estimated number of instructions per invocation; 450 is the default; -1 if not known.</td>
</tr>
<tr>
<td>IOS_PER_ARGBYTE</td>
<td>DOUBLE</td>
<td>Y</td>
<td></td>
<td>Estimated number of I/Os per input argument byte; 0 is the default; -1 if not known.</td>
</tr>
<tr>
<td>INSTS_PER_ARGBYTE</td>
<td>DOUBLE</td>
<td>Y</td>
<td></td>
<td>Estimated number of instructions per input argument byte; 0 is the default; -1 if not known.</td>
</tr>
<tr>
<td>PERCENT_ARGBYTES</td>
<td>SMALLINT</td>
<td>Y</td>
<td></td>
<td>Estimated average percent of input argument bytes that the routine will actually read; 100 is the default; -1 if not known.</td>
</tr>
<tr>
<td>INITIAL_IOS</td>
<td>DOUBLE</td>
<td>Y</td>
<td></td>
<td>Estimated number of I/Os performed the first time that the routine is invoked; 0 is the default; -1 if not known.</td>
</tr>
<tr>
<td>INITIAL_INSTS</td>
<td>DOUBLE</td>
<td>Y</td>
<td></td>
<td>Estimated number of instructions executed the first time the routine is invoked; 0 is the default; -1 if not known.</td>
</tr>
<tr>
<td>CARDINALITY</td>
<td>BIGINT</td>
<td>Y</td>
<td></td>
<td>Predicted cardinality of a table function; -1 if not known, or if the routine is not a table function.</td>
</tr>
<tr>
<td>SELECTIVITY</td>
<td>DOUBLE</td>
<td>Y</td>
<td></td>
<td>For user-defined predicates; -1 if there are no user-defined predicates.</td>
</tr>
</tbody>
</table>
SYSSTAT.TABLES

Each row represents a table, view, alias, or nickname. Each table or view hierarchy has one additional row representing the hierarchy table or hierarchy view that implements the hierarchy. Catalog tables and views are included.

Table 216. SYSSTAT.TABLES Catalog View

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Updat-able</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TABSCHEMA</td>
<td>VARCHAR (128)</td>
<td></td>
<td></td>
<td>Schema name of the object.</td>
</tr>
<tr>
<td>TABNAME</td>
<td>VARCHAR (128)</td>
<td></td>
<td></td>
<td>Unqualified name of the object.</td>
</tr>
<tr>
<td>CARD</td>
<td>BIGINT</td>
<td></td>
<td>Y</td>
<td>Total number of rows in the table; -1 if statistics are not collected.</td>
</tr>
<tr>
<td>NPAGES</td>
<td>BIGINT</td>
<td></td>
<td>Y</td>
<td>Total number of pages on which the rows of the table exist; -1 for a view or alias, or if statistics are not collected; -2 for a subtable or hierarchy table.</td>
</tr>
<tr>
<td>FPAGES</td>
<td>BIGINT</td>
<td></td>
<td>Y</td>
<td>Total number of pages; -1 for a view or alias, or if statistics are not collected; -2 for a subtable or hierarchy table.</td>
</tr>
<tr>
<td>OVERFLOW</td>
<td>BIGINT</td>
<td></td>
<td>Y</td>
<td>Total number of overflow records in the table; -1 for a view or alias, or if statistics are not collected; -2 for a subtable or hierarchy table.</td>
</tr>
</tbody>
</table>
| CLUSTERED    | CHAR (1)      |          | Y          | • Y = Table is multidimensionally clustered (even if only by one dimension)  
• Null value = Table is not multidimensionally clustered |
| ACTIVE_BLOCKS| BIGINT        |          | Y          | Total number of active blocks in the table, or -1. Applies to multidimensional clustering (MDC) tables only. |
| AVGCOMPRESSEDROWSIZE | SMALLINT |          | Y          | Average length (in bytes) of compressed rows in this table; -1 if statistics are not collected. |
| AVGROWCOMPRESSIONRATIO | REAL     |          | Y          | For compressed rows in the table, this is the average compression ratio by row; that is, the average uncompressed row length divided by the average compressed row length; -1 if statistics are not collected. |
| AVGROWSIZE   | SMALLINT      |          |            | Average length (in bytes) of both compressed and uncompressed rows in this table; -1 if statistics are not collected. |
| PCTROWSCOMPRESSED | REAL  |          | Y          | Compressed rows as a percentage of the total number of rows in the table; -1 if statistics are not collected. |
| PCTPAGESAVED | SMALLINT      |          | Y          | Approximate percentage of pages saved in the table as a result of row compression. This value includes overhead bytes for each user data row in the table, but does not include the space that is consumed by dictionary overhead; -1 if statistics are not collected. |
SYSSTAT.TABLES
Appendix E. Federated systems
Valid server types in SQL statements

Server types indicate the kind of data source that the server definition represents.

Server types vary by vendor, purpose, and operating system. Supported values depend on the data source.

For most data sources, you must specify a valid server type in the CREATE SERVER statement.

*Table 217. Data sources and server types*

<table>
<thead>
<tr>
<th>Data source</th>
<th>Server type</th>
</tr>
</thead>
<tbody>
<tr>
<td>BioRS</td>
<td>A server type is not required in the CREATE SERVER statement.</td>
</tr>
<tr>
<td>Excel</td>
<td>A server type is not required in the CREATE SERVER statement.</td>
</tr>
<tr>
<td>IBM DB2 Universal Database for Linux, UNIX, and Windows</td>
<td>DB2/UDB</td>
</tr>
<tr>
<td>IBM DB2 Universal Database for System i and AS/400®</td>
<td>DB2/ISERIES</td>
</tr>
<tr>
<td>IBM DB2 Universal Database for z/OS</td>
<td>DB2/ZOS</td>
</tr>
<tr>
<td>IBM DB2 for VM</td>
<td>DB2/VM</td>
</tr>
<tr>
<td>Informix</td>
<td>INFORMIX</td>
</tr>
<tr>
<td>JDBC</td>
<td>JDBC (Required for JDBC data sources that are supported by JDBC drivers 3.0 and later.)</td>
</tr>
<tr>
<td>Microsoft SQL Server</td>
<td>MSSQLSERVER (Required for data sources supported by the DataDirect Connect ODBC 4.2 (or later) driver or the Microsoft SQL Server ODBC 3.0 (or later) driver.)</td>
</tr>
<tr>
<td>ODBC</td>
<td>ODBC (Required for ODBC data sources that are supported by the ODBC 3.x driver.)</td>
</tr>
<tr>
<td>OLE DB</td>
<td>A server type is not required in the CREATE SERVER statement.</td>
</tr>
<tr>
<td>Oracle</td>
<td>ORACLE (Required for Oracle data sources supported by Oracle NET8 client software.)</td>
</tr>
<tr>
<td>Sybase (CTLIB)</td>
<td>SYBASE</td>
</tr>
<tr>
<td>Table-structured files</td>
<td>A server type is not required in the CREATE SERVER statement.</td>
</tr>
<tr>
<td>Teradata</td>
<td>TERADATA</td>
</tr>
<tr>
<td>Web services</td>
<td>A server type is not required in the CREATE SERVER statement.</td>
</tr>
<tr>
<td>XML</td>
<td>A server type is not required in the CREATE SERVER statement.</td>
</tr>
</tbody>
</table>
Function mapping options for federated systems

The federated server provides default mappings between DB2 functions and data source functions. For most data sources, the default function mappings are in the wrappers. To use a data source function that the federated server does not recognize or to change the default mapping, you create a function mapping.

When you create a function mapping, you specify the name of the data source function and must enable the mapped function. Then when you use the mapped function, the query optimizer compares the cost of running the function at the data source with the cost of running the function at the federated server.

Table 218. Options for function mappings

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DISABLE</td>
<td>Enable or disable a default function mapping. Valid values are Y and N. The default is N.</td>
</tr>
<tr>
<td>REMOTE_NAME</td>
<td>The name of the data source function. The default is the local name.</td>
</tr>
</tbody>
</table>
Default forward data type mappings

The two kinds of mappings between data source data types and federated database data types are forward type mappings and reverse type mappings. In a forward type mapping, the mapping is from a remote type to a comparable local type.

You can override a default type mapping, or create a new type mapping with the CREATE TYPE MAPPING statement.

These mappings are valid with all the supported versions, unless otherwise noted.

For all default forward data types mapping from a data source to the federated database, the federated schema is SYSIBM.

The following tables show the default forward mappings between federated database data types and data source data types.
Default forward data type mappings for DB2 Database for Linux, UNIX, and Windows data sources

The following table lists the default forward data type mappings for DB2 Database for Linux, UNIX, and Windows data sources.

<table>
<thead>
<tr>
<th>Remote Typename</th>
<th>Remote Lower Len</th>
<th>Remote Upper Len</th>
<th>Remote Lower Scale</th>
<th>Remote Upper Scale</th>
<th>Remote Bit Data</th>
<th>Remote Data Operators</th>
<th>Federated Typename</th>
<th>Federated Length</th>
<th>Federated Scale</th>
<th>Federated Bit Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIGINT</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>BIGINT</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>BLOB</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>BLOB</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>CHAR</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>CHAR</td>
<td>0</td>
<td>N</td>
<td>-</td>
</tr>
<tr>
<td>CHAR</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Y</td>
<td>CHAR</td>
<td>0</td>
<td>Y</td>
<td>-</td>
</tr>
<tr>
<td>CLOB</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Y</td>
<td>-</td>
<td>-</td>
<td>CLOB</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DATE</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>DATE</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DATE</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>TIMESTAMP</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DBCLOB</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>DBCLOB</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>DECIMAL</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DECFLOAT</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>DECFLOAT</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>DOUBLE</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>FLOAT</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>FLOAT</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>GRAPHIC</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>GRAPHIC</td>
<td>0</td>
<td>N</td>
<td>-</td>
</tr>
<tr>
<td>INTEGER</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>INTEGER</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>LONGVAR</td>
<td>-</td>
<td>-</td>
<td>N</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>CLOB</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>LONGVAR</td>
<td>-</td>
<td>-</td>
<td>Y</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>BLOB</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>LONGVARG</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>DBCLOB</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>REAL</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>REAL</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>SMALLINT</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>TIME</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>TIME</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>TIMESTAMP(p)</td>
<td>-</td>
<td>p</td>
<td>p</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>TIMESTAMP(p)</td>
<td>p</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>VARCHAR</td>
<td>0</td>
<td>N</td>
<td>-</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>-</td>
<td>-</td>
<td>Y</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>VARCHAR</td>
<td>0</td>
<td>Y</td>
<td>-</td>
</tr>
<tr>
<td>VARGRAPH</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>VARGRAPH</td>
<td>0</td>
<td>N</td>
<td>-</td>
</tr>
<tr>
<td>VARGRAPHIC</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>VARGRAPHIC</td>
<td>0</td>
<td>N</td>
<td>-</td>
</tr>
</tbody>
</table>

Note:
1. The federated type is TIMESTAMP(0) if the date_compat configuration parameter is set to ON.
2. The SAME_DECFLT_ROUNDING server option is set to N by default and operations will not be pushed down to the remote data source unless SAME_DECFLT_ROUNDING is set to Y. For information on the SAME_DECFLT_ROUNDING server option, see [DB2 database options reference](#).
Default forward data type mappings for DB2 for System i data sources

The following table lists the default forward data type mappings for DB2 for iSeries® data sources.

Table 220. DB2 for System i default forward data type mappings (Not all columns shown)

<table>
<thead>
<tr>
<th>Remote Typename</th>
<th>Remote Lower Len</th>
<th>Remote Upper Len</th>
<th>Remote Lower Scale</th>
<th>Remote Upper Scale</th>
<th>Remote Bit Data</th>
<th>Remote Data Operators</th>
<th>Federated Typename</th>
<th>Federated Length</th>
<th>Federated Scale</th>
<th>Federated Bit Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLOB</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>BLOB</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>CHAR</td>
<td>1</td>
<td>254</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>CHAR</td>
<td>0</td>
<td>N</td>
<td>-</td>
</tr>
<tr>
<td>CHAR</td>
<td>255</td>
<td>32762</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>VARCHAR</td>
<td>0</td>
<td>N</td>
<td>-</td>
</tr>
<tr>
<td>CHAR</td>
<td>1</td>
<td>254</td>
<td>-</td>
<td>Y</td>
<td>-</td>
<td>-</td>
<td>CHAR</td>
<td>0</td>
<td>Y</td>
<td>-</td>
</tr>
<tr>
<td>CHAR</td>
<td>255</td>
<td>32762</td>
<td>-</td>
<td>Y</td>
<td>-</td>
<td>-</td>
<td>VARCHAR</td>
<td>0</td>
<td>Y</td>
<td>-</td>
</tr>
<tr>
<td>CLOB</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>CLOB</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DATE</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>DATE</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DBCLOB</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>DBCLOB</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>DECIMAL</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>FLOAT</td>
<td>4</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>REAL</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>FLOAT</td>
<td>8</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>DOUBLE</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>GRAPHIC</td>
<td>1</td>
<td>127</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>GRAPHIC</td>
<td>0</td>
<td>N</td>
<td>-</td>
</tr>
<tr>
<td>GRAPHIC</td>
<td>128</td>
<td>16336</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>VARGRAPHIC</td>
<td>0</td>
<td>N</td>
<td>-</td>
</tr>
<tr>
<td>INTEGER</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>INTEGER</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>NUMERIC</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>DECIMAL</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>SMALLINT</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>TIME</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>TIME</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>TIMESTAMP(6)</td>
<td>6</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>1</td>
<td>32762</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>VARCHAR</td>
<td>0</td>
<td>N</td>
<td>-</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>1</td>
<td>32762</td>
<td>-</td>
<td>Y</td>
<td>-</td>
<td>-</td>
<td>VARCHAR</td>
<td>0</td>
<td>Y</td>
<td>-</td>
</tr>
<tr>
<td>VARG</td>
<td>1</td>
<td>16336</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>VARGRAPHIC</td>
<td>0</td>
<td>N</td>
<td>-</td>
</tr>
<tr>
<td>VARGRAPHIC</td>
<td>1</td>
<td>16336</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>VARGRAPHIC</td>
<td>0</td>
<td>N</td>
<td>-</td>
</tr>
</tbody>
</table>
Default forward data type mappings for DB2 for VM and VSE data sources

The following table lists the default forward data type mappings for DB2 for VM and VSE data sources.

### Table 221. DB2 Server for VM and VSE default forward data type mappings (Not all columns shown)

<table>
<thead>
<tr>
<th>Remote Typename</th>
<th>Remote Lower Len</th>
<th>Remote Upper Len</th>
<th>Remote Lower Scale</th>
<th>Remote Upper Scale</th>
<th>Remote Bit Data</th>
<th>Remote Data Operators</th>
<th>Federated Typename</th>
<th>Federated Length</th>
<th>Federated Scale</th>
<th>Federated Bit Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLOB</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>BLOB</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>CHAR</td>
<td>1</td>
<td>254</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>CHAR</td>
<td>-</td>
<td>0</td>
<td>N</td>
</tr>
<tr>
<td>CHAR</td>
<td>1</td>
<td>254</td>
<td>-</td>
<td>Y</td>
<td>-</td>
<td>-</td>
<td>CHAR</td>
<td>-</td>
<td>0</td>
<td>Y</td>
</tr>
<tr>
<td>CLOB</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>CLOB</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DATE</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>DATE</td>
<td>-</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>DBAHW</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>SMALLINT</td>
<td>-</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>DBAINT</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>INTEGER</td>
<td>-</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>DBCLOB</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>DBCLOB</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>DECIMAL</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>FLOAT</td>
<td>4</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>REAL</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>FLOAT</td>
<td>8</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>DOUBLE</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>GRAPHIC</td>
<td>1</td>
<td>127</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>GRAPHIC</td>
<td>-</td>
<td>0</td>
<td>N</td>
</tr>
<tr>
<td>INTEGER</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>INTEGER</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>SMALLINT</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>TIME</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>TIME</td>
<td>-</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>TIMESTAMP(6)</td>
<td>-</td>
<td>6</td>
<td>-</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>1</td>
<td>32672</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>VARCHAR</td>
<td>-</td>
<td>0</td>
<td>N</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>1</td>
<td>32672</td>
<td>-</td>
<td>Y</td>
<td>-</td>
<td>-</td>
<td>VARCHAR</td>
<td>-</td>
<td>0</td>
<td>Y</td>
</tr>
<tr>
<td>VARGRAPHIC</td>
<td>1</td>
<td>16336</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>VARGRAPHIC</td>
<td>-</td>
<td>0</td>
<td>N</td>
</tr>
<tr>
<td>VARGraph</td>
<td>1</td>
<td>16336</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>VARGRAPHIC</td>
<td>-</td>
<td>0</td>
<td>N</td>
</tr>
</tbody>
</table>
**Default forward data type mappings for DB2 for z/OS data sources**

The following table lists the default forward data type mappings for DB2 for z/OS data sources.

<table>
<thead>
<tr>
<th>Remote Typename</th>
<th>Remote Lower Len</th>
<th>Remote Upper Scale</th>
<th>Remote Lower Len</th>
<th>Remote Upper Scale</th>
<th>Remote Lower Scale</th>
<th>Remote Upper Scale</th>
<th>Remote Bit Data</th>
<th>Remote Data Operators</th>
<th>Federated Typename</th>
<th>Federated Length</th>
<th>Federated Scale</th>
<th>Federated Bit Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLOB</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>BLOB</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>CHAR</td>
<td>1</td>
<td>254</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>CHAR</td>
<td>CHAR</td>
<td>-</td>
<td>0</td>
<td>N</td>
</tr>
<tr>
<td>CHAR</td>
<td>255</td>
<td>32672</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>VARCHAR</td>
<td>VARCHAR</td>
<td>-</td>
<td>0</td>
<td>N</td>
</tr>
<tr>
<td>CHAR</td>
<td>1</td>
<td>254</td>
<td>-</td>
<td>Y</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>CHAR</td>
<td>CHAR</td>
<td>-</td>
<td>0</td>
<td>Y</td>
</tr>
<tr>
<td>CHAR</td>
<td>255</td>
<td>32672</td>
<td>-</td>
<td>Y</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>VARCHAR</td>
<td>VARCHAR</td>
<td>-</td>
<td>0</td>
<td>Y</td>
</tr>
<tr>
<td>CLOB</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>CLOB</td>
<td>CLOB</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DATE</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>DATE</td>
<td>DATE</td>
<td>-</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>DBCLOB</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>DBCLOB</td>
<td>DBCLOB</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>DECIMAL</td>
<td>DECIMAL</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>FLOAT</td>
<td>4</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>REAL</td>
<td>REAL</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>FLOAT</td>
<td>8</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>DOUBLE</td>
<td>DOUBLE</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>GRAPHIC</td>
<td>1</td>
<td>127</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>GRAPHIC</td>
<td>GRAPHIC</td>
<td>-</td>
<td>0</td>
<td>N</td>
</tr>
<tr>
<td>INTEGER</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>INTEGER</td>
<td>INTEGER</td>
<td>-</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>ROWID</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Y</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>VARCHAR</td>
<td>VARCHAR</td>
<td>40</td>
<td>-</td>
<td>Y</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>SMALLINT</td>
<td>SMALLINT</td>
<td>-</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>TIME</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>TIME</td>
<td>TIME</td>
<td>-</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>TIMESTAMP(6)</td>
<td>TIMESTAMP(6)</td>
<td>-</td>
<td>6</td>
<td>-</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>1</td>
<td>32672</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>VARCHAR</td>
<td>VARCHAR</td>
<td>-</td>
<td>0</td>
<td>N</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>1</td>
<td>32672</td>
<td>-</td>
<td>Y</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>VARCHAR</td>
<td>VARCHAR</td>
<td>-</td>
<td>0</td>
<td>Y</td>
</tr>
<tr>
<td>VARG</td>
<td>1</td>
<td>16336</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>VARGRAPHIC</td>
<td>VARGRAPHIC</td>
<td>-</td>
<td>0</td>
<td>N</td>
</tr>
<tr>
<td>VARGRAPHIC</td>
<td>1</td>
<td>16336</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>VARGRAPHIC</td>
<td>VARGRAPHIC</td>
<td>-</td>
<td>0</td>
<td>N</td>
</tr>
</tbody>
</table>
## Default forward data type mappings for Informix data sources

The following table lists the default forward data type mappings for Informix data sources.

<table>
<thead>
<tr>
<th>Remote Typename</th>
<th>Remote Lower Len</th>
<th>Remote Upper Len</th>
<th>Remote Lower Scale</th>
<th>Remote Upper Scale</th>
<th>Remote Bit Data</th>
<th>Remote Data Operators</th>
<th>Federated Typename</th>
<th>Federated Length</th>
<th>Federated Scale</th>
<th>Federated Bit Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLOB</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>BLOB</td>
<td>2147483647</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>BOOLEAN</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>CHARACTER</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>BYTE</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>BLOB</td>
<td>2147483647</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>CHAR</td>
<td>1</td>
<td>254</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>CHARACTER</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>CHAR</td>
<td>255</td>
<td>32672</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>VARCHAR</td>
<td>2147483647</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>CLOB</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>CLOB</td>
<td>2147483647</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DATE</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>DATE</td>
<td>4</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DATE</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>TIMESTAMP</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DATETIME</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>4</td>
<td>-</td>
<td>-</td>
<td>DATE</td>
<td>4</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DATETIME</td>
<td>1</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>-</td>
<td>-</td>
<td>TIME</td>
<td>3</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DATETIME</td>
<td>2</td>
<td>11</td>
<td>15</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>TIMESTAMP(4)</td>
<td>10</td>
<td>6</td>
<td>-</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>1</td>
<td>31</td>
<td>0</td>
<td>31</td>
<td>-</td>
<td>-</td>
<td>DECIMAL</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>32</td>
<td>130</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>DOUBLE</td>
<td>8</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>1</td>
<td>32</td>
<td>255</td>
<td>255</td>
<td>-</td>
<td>-</td>
<td>DOUBLE</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>FLOAT</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>DOUBLE</td>
<td>8</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>INTEGER</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>INTEGER</td>
<td>4</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>INTERVAL</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>VARCHAR</td>
<td>25</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>INT8</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>BIGINT</td>
<td>19</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>LVARCHAR</td>
<td>1</td>
<td>32672</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>VARCHAR</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>MONEY</td>
<td>1</td>
<td>31</td>
<td>0</td>
<td>31</td>
<td>-</td>
<td>-</td>
<td>DECIMAL</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>MONEY</td>
<td>32</td>
<td>32</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>DOUBLE</td>
<td>8</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>NCHAR</td>
<td>1</td>
<td>254</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>CHARACTER</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>NVARCHAR</td>
<td>1</td>
<td>32672</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>VARCHAR</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>REAL</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>REAL</td>
<td>4</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SERIAL</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>INTEGER</td>
<td>4</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SERIAL8</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>BIGINT</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SMALLFLOAT</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>REAL</td>
<td>4</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>SMALLINT</td>
<td>2</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>TEXT</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>CLOB</td>
<td>2147483647</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>1</td>
<td>32672</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>VARCHAR</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

### Notes:

1. The federated type is TIMESTAMP(0) if the date_compatible configuration parameter is set to ON.
2. For the Informix DATETIME data type, the DB2 UNIX and Windows federated server uses the Informix high-level qualifier as the REMOTE_LENGTH and the Informix low-level qualifier as the REMOTE_SCALE.

The Informix qualifiers are the “TU_” constants defined in the Informix Client SDK datatime.h file. The constants are:

- 0 = YEAR
- 8 = MINUTE
- 13 = FRACTION(3)
- 2 = MONTH
- 10 = SECOND
- 14 = FRACTION(4)
- 4 = DAY
- 11 = FRACTION(1)
- 15 = FRACTION(5)
Default forward data type mappings for Informix data sources

Table 223. Informix default forward data type mappings (Not all columns shown) (continued)

<table>
<thead>
<tr>
<th>Remote Typename</th>
<th>Remote Lower Len</th>
<th>Remote Upper Len</th>
<th>Remote Lower Scale</th>
<th>Remote Upper Scale</th>
<th>Remote Bit Data</th>
<th>Federated Typename</th>
<th>Federated Length</th>
<th>Federated Scale</th>
<th>Federated Bit Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 = HOUR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12 = FRACTION(2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Default forward data type mappings for Microsoft SQL Server data sources

The following table lists the default forward data type mappings for Microsoft SQL Server data sources.

<table>
<thead>
<tr>
<th>Remote Typename</th>
<th>Remote Lower Len</th>
<th>Remote Upper Len</th>
<th>Remote Lower Scale</th>
<th>Remote Upper Scale</th>
<th>Remote Bit Data</th>
<th>Remote Data Operators</th>
<th>Federated Typename</th>
<th>Federated Length</th>
<th>Federated Scale</th>
<th>Federated Bit Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>bigint 1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>BIGINT</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>binary</td>
<td>1 254</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>CHARACTER</td>
<td>-</td>
<td>Y</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>binary</td>
<td>255 8000</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>VARCHAR</td>
<td>-</td>
<td>-</td>
<td>Y</td>
<td>-</td>
</tr>
<tr>
<td>bit</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>SMALLINT</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>char</td>
<td>1 254</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>CHAR</td>
<td>-</td>
<td>N</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>char</td>
<td>255 8000</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>VARCHAR</td>
<td>-</td>
<td>N</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>datetime</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>TIMESTAMP(6)</td>
<td>10</td>
<td>6</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>decimal</td>
<td>1 31 0 31</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>DECIMAL</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>decimal</td>
<td>32 38 0 38</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>DOUBLE</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>float</td>
<td>- 8</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>DOUBLE</td>
<td>8</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>float</td>
<td>- 4</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>REAL</td>
<td>4</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>image</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>BLOB</td>
<td>2147483647</td>
<td>Y</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>int</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>INTEGER</td>
<td>4</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>money</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>DECIMAL</td>
<td>19</td>
<td>4</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>nchar</td>
<td>1 127</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>CHAR</td>
<td>-</td>
<td>N</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>nchar</td>
<td>128 4000</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>VARCHAR</td>
<td>-</td>
<td>N</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>numeric</td>
<td>1 31 0 31</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>DECIMAL</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>numeric</td>
<td>32 38 0 38</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>DOUBLE</td>
<td>8</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>ntext</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>CLOB</td>
<td>2147483647</td>
<td>Y</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>nvarchar</td>
<td>1 4000</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>VARCHAR</td>
<td>-</td>
<td>N</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>real</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>REAL</td>
<td>4</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>smallint</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>SMALLINT</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>smalldatetime</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>TIMESTAMP(6)</td>
<td>10</td>
<td>6</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>smallmoney</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>DECIMAL</td>
<td>10</td>
<td>4</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SQL_BIGINT</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>BIGINT</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SQL_BINARY</td>
<td>1 254</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>CHARACTER</td>
<td>-</td>
<td>Y</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SQL_BINARY</td>
<td>255 8000</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>VARCHAR</td>
<td>-</td>
<td>Y</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SQL_BIT</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>SMALLINT</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SQL_CHAR</td>
<td>1 254</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>CHAR</td>
<td>-</td>
<td>N</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SQL_CHAR</td>
<td>255 8000</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>VARCHAR</td>
<td>-</td>
<td>N</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SQL_DATE</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>DATE</td>
<td>4</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SQL_DECIMAL</td>
<td>1 31 0 31</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>DECIMAL</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SQL_DECIMAL</td>
<td>32 38 0 38</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>DOUBLE</td>
<td>8</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SQL_DOUBLE</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>DOUBLE</td>
<td>8</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SQL_FLOAT</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>DOUBLE</td>
<td>8</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SQL_GUID</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>VARCHAR</td>
<td>-</td>
<td>Y</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SQL_INTEGER</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>INTEGER</td>
<td>4</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SQL_LONGVARCHAR</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>CLOB</td>
<td>2147483647</td>
<td>N</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
## Default forward data type mappings for Microsoft SQL Server data sources

**Table 224. Microsoft SQL Server default forward data type mappings (continued)**

<table>
<thead>
<tr>
<th>Remote Typename</th>
<th>Remote Lower Len</th>
<th>Remote Upper Len</th>
<th>Remote Lower Scale</th>
<th>Remote Upper Scale</th>
<th>Remote Bit Data</th>
<th>Remote Data Operators</th>
<th>Federated Typename</th>
<th>Federated Length</th>
<th>Federated Scale</th>
<th>Federated Bit Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_LONGVARBINARY</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>BLOB</td>
<td>-</td>
<td>-</td>
<td>Y</td>
</tr>
<tr>
<td>SQL_NUMERIC</td>
<td>1</td>
<td>31</td>
<td>0</td>
<td>31</td>
<td>-</td>
<td>-</td>
<td>DECIMAL</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SQL_NUMERIC</td>
<td>32</td>
<td>38</td>
<td>0</td>
<td>38</td>
<td>-</td>
<td>-</td>
<td>DOUBLE</td>
<td>8</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SQL_REAL</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>REAL</td>
<td>8</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SQL_SMALLINT</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>SMALLINT</td>
<td>2</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SQL_TIME</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>TIME</td>
<td>3</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SQL_TIMESTAMP</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>TIMESTAMP</td>
<td>10</td>
<td>6</td>
<td>-</td>
</tr>
<tr>
<td>SQL_TINYINT</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>SMALLINT</td>
<td>2</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SQL_VARBINARY</td>
<td>1</td>
<td>8000</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>VARCHAR</td>
<td>-</td>
<td>-</td>
<td>Y</td>
</tr>
<tr>
<td>SQL_VARCHAR</td>
<td>1</td>
<td>8000</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>VARCHAR</td>
<td>-</td>
<td>-</td>
<td>N</td>
</tr>
<tr>
<td>SQL_WCHAR</td>
<td>1</td>
<td>254</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>CHARACTER</td>
<td>-</td>
<td>-</td>
<td>N</td>
</tr>
<tr>
<td>SQL_WCHAR</td>
<td>255</td>
<td>8800</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>VARCHAR</td>
<td>-</td>
<td>-</td>
<td>N</td>
</tr>
<tr>
<td>SQL_WLONGVARCHAR</td>
<td>1073741823</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>CLOB</td>
<td>2147483647</td>
<td>-</td>
<td>N</td>
</tr>
<tr>
<td>SQL_WVARCHAR</td>
<td>1</td>
<td>16336</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>VARCHAR</td>
<td>-</td>
<td>-</td>
<td>N</td>
</tr>
<tr>
<td>text</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>CLOB</td>
<td>-</td>
<td>-</td>
<td>N</td>
</tr>
<tr>
<td>timestamp</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>VARCHAR</td>
<td>8</td>
<td>-</td>
<td>Y</td>
</tr>
<tr>
<td>tinyint</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>SMALLINT</td>
<td>2</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>uniqueidentifier</td>
<td>1</td>
<td>4000</td>
<td>-</td>
<td>-</td>
<td>Y</td>
<td>-</td>
<td>VARCHAR</td>
<td>16</td>
<td>-</td>
<td>Y</td>
</tr>
<tr>
<td>varbinary</td>
<td>1</td>
<td>8000</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>VARCHAR</td>
<td>-</td>
<td>-</td>
<td>Y</td>
</tr>
<tr>
<td>varchar</td>
<td>1</td>
<td>8000</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>VARCHAR</td>
<td>-</td>
<td>-</td>
<td>N</td>
</tr>
</tbody>
</table>

**Note:**
1. This type mapping is valid only with Microsoft SQL Server Version 2000.
The following table lists the default forward data type mappings for ODBC data sources.

<table>
<thead>
<tr>
<th>Remote Typename</th>
<th>Remote Lower Len</th>
<th>Remote Upper Len</th>
<th>Remote Lower Scale</th>
<th>Remote Upper Scale</th>
<th>Remote Bit Data</th>
<th>Remote Data Operators</th>
<th>Federated Typename</th>
<th>Federated Length</th>
<th>Federated Scale</th>
<th>Federated Bit Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_BIGINT</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>BIGINT</td>
<td>8</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SQL_BINARY</td>
<td>1</td>
<td>254</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>CHARACTER</td>
<td>-</td>
<td>Y</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SQL_BINARY</td>
<td>255</td>
<td>32767</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>VARCHAR</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SQL_BIT</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>SMALLINT</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SQL_CHAR</td>
<td>1</td>
<td>254</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>CHAR</td>
<td>-</td>
<td>N</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SQL_CHAR</td>
<td>255</td>
<td>32767</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>VARCHAR</td>
<td>-</td>
<td>N</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SQL_DATE</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>DATE</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SQL_DATE</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>DATE</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SQL_DECIMAL</td>
<td>1</td>
<td>31</td>
<td>0</td>
<td>31</td>
<td>-</td>
<td>DECIMAL</td>
<td>8</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SQL_DECIMAL</td>
<td>32</td>
<td>38</td>
<td>0</td>
<td>38</td>
<td>-</td>
<td>DOUBLE</td>
<td>8</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SQL_DOUBLE</td>
<td>8</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>FLOAT</td>
<td>8</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SQL_FLOAT</td>
<td>4</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>FLOAT</td>
<td>4</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SQL_INTEGER</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>INTEGER</td>
<td>4</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SQL_LONGVARCHAR</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>CLOB</td>
<td>2147483647</td>
<td>N</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SQL_LONGVARBINARY</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>BLOB</td>
<td>2147483647</td>
<td>Y</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SQL_NUMERIC</td>
<td>1</td>
<td>31</td>
<td>0</td>
<td>31</td>
<td>-</td>
<td>DECIMAL</td>
<td>8</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SQL_NUMERIC</td>
<td>32</td>
<td>32</td>
<td>0</td>
<td>31</td>
<td>-</td>
<td>DOUBLE</td>
<td>8</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SQL_REAL</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>REAL</td>
<td>4</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SQL_SMALLINT</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>SMALLINT</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SQL_TIMESTAMP</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>TIMESTAMP</td>
<td>6</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SQL_TIMESTAMP</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>TIMESTAMP</td>
<td>6</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SQL_TYPE_DATE</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>DATE</td>
<td>4</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SQL_TYPE_TIME</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>TIME</td>
<td>3</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SQL_TYPE_TIMESTAMP</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>TIMESTAMP</td>
<td>10</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SQL_TINYINT</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>SMALLINT</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SQL_VARBINARY</td>
<td>1</td>
<td>32767</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>VARCHAR</td>
<td>-</td>
<td>Y</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SQL_VARCHAR</td>
<td>1</td>
<td>32767</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>VARCHAR</td>
<td>-</td>
<td>N</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SQL_WCHAR</td>
<td>1</td>
<td>127</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>CHAR</td>
<td>-</td>
<td>N</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SQL_WCHAR</td>
<td>128</td>
<td>16384</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>VARCHAR</td>
<td>-</td>
<td>N</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SQL_WVARCHAR</td>
<td>1</td>
<td>16384</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>VARCHAR</td>
<td>-</td>
<td>N</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SQL_WLONGVARCHAR</td>
<td>1073741823</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>CLOB</td>
<td>2147483647</td>
<td>N</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Note:
1. The federated type is TIMESTAMP(0) if the date_compat configuration parameter is set to ON.
### Default forward data type mappings for Oracle NET8 data sources

The following table lists the default forward data type mappings for Oracle NET8 data sources.

<table>
<thead>
<tr>
<th>Remote Type</th>
<th>Remote Lower Lim</th>
<th>Remote Upper Lim</th>
<th>Remote Lower Scale</th>
<th>Remote Upper Scale</th>
<th>Remote Bit Data</th>
<th>Federation Typename</th>
<th>Federation Length</th>
<th>Federation Scale</th>
<th>Federation Bit Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLOB</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>BLOB</td>
<td>2147483647</td>
<td>0</td>
<td>Y</td>
</tr>
<tr>
<td>CHAR</td>
<td>1</td>
<td>254</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>CHAR</td>
<td>0</td>
<td>0</td>
<td>N</td>
</tr>
<tr>
<td>CHAR</td>
<td>255</td>
<td>2000</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>VARCHAR</td>
<td>0</td>
<td>0</td>
<td>N</td>
</tr>
<tr>
<td>CLOB</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>CLOB</td>
<td>2147483647</td>
<td>0</td>
<td>N</td>
</tr>
<tr>
<td>DATE</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>TIMESTAMP(6)</td>
<td>0</td>
<td>0</td>
<td>N</td>
</tr>
<tr>
<td>FLOAT</td>
<td>1</td>
<td>126</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>DOUBLE</td>
<td>0</td>
<td>0</td>
<td>N</td>
</tr>
<tr>
<td>LONG</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>CLOB</td>
<td>2147483647</td>
<td>0</td>
<td>N</td>
</tr>
<tr>
<td>LONG RAW</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>BLOB</td>
<td>2147483647</td>
<td>0</td>
<td>Y</td>
</tr>
<tr>
<td>NUMBER</td>
<td>10</td>
<td>18</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>BIGINT</td>
<td>0</td>
<td>0</td>
<td>N</td>
</tr>
<tr>
<td>NUMBER</td>
<td>1</td>
<td>38</td>
<td>-84</td>
<td>127</td>
<td>-</td>
<td>DOUBLE</td>
<td>0</td>
<td>0</td>
<td>N</td>
</tr>
<tr>
<td>NUMBER</td>
<td>1</td>
<td>31</td>
<td>0</td>
<td>31</td>
<td>-</td>
<td>DECIMAL</td>
<td>0</td>
<td>0</td>
<td>N</td>
</tr>
<tr>
<td>NUMBER</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>SMALLINT</td>
<td>0</td>
<td>0</td>
<td>N</td>
</tr>
<tr>
<td>NUMBER</td>
<td>5</td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>INTEGER</td>
<td>0</td>
<td>0</td>
<td>N</td>
</tr>
<tr>
<td>NUMBER</td>
<td>-</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>DECIMAL</td>
<td>0</td>
<td>0</td>
<td>N</td>
</tr>
<tr>
<td>RAW</td>
<td>1</td>
<td>2000</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>VARCHAR</td>
<td>0</td>
<td>0</td>
<td>Y</td>
</tr>
<tr>
<td>ROWID</td>
<td>0</td>
<td>0</td>
<td>NULL</td>
<td>-</td>
<td>-</td>
<td>CHAR</td>
<td>18</td>
<td>0</td>
<td>N</td>
</tr>
<tr>
<td>TIMESTAMP(p)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>TIMESTAMP(6)</td>
<td>10</td>
<td>6</td>
<td>N</td>
</tr>
<tr>
<td>VARCHAR2</td>
<td>1</td>
<td>4000</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>VARCHAR</td>
<td>0</td>
<td>0</td>
<td>N</td>
</tr>
</tbody>
</table>

Note:

1. TIMESTAMP(p) is a timestamp with a variable scale from 0-9. The scale of the Oracle timestamp is mapped to TIMESTAMP(6) by default. You can change this default type mapping and map the Oracle TIMESTAMP to a federated TIMESTAMP of the same scale by using a user-defined type mapping.
2. This type mapping is valid only for Oracle 9i (or later) client and server configurations.
Default forward data type mappings for Sybase data sources

The following table lists the default forward data type mappings for Sybase data sources.

<table>
<thead>
<tr>
<th>Remote Typename</th>
<th>Remote Lower Len</th>
<th>Remote Upper Len</th>
<th>Remote Lower Scale</th>
<th>Remote Upper Scale</th>
<th>Remote Bit Data</th>
<th>Remote Data Operators</th>
<th>Federated Typename</th>
<th>Federated Length</th>
<th>Federated Scale</th>
<th>Federated Bit Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>binary</td>
<td>1</td>
<td>254</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>CHAR</td>
<td>-</td>
<td>-</td>
<td>Y</td>
</tr>
<tr>
<td>binary</td>
<td>255</td>
<td>32672</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>VARCHAR</td>
<td>-</td>
<td>-</td>
<td>Y</td>
</tr>
<tr>
<td>bit</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>SMALLINT</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>char</td>
<td>1</td>
<td>254</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>CHAR</td>
<td>-</td>
<td>-</td>
<td>N</td>
</tr>
<tr>
<td>char</td>
<td>255</td>
<td>32672</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>VARCHAR</td>
<td>-</td>
<td>-</td>
<td>N</td>
</tr>
<tr>
<td>char null (see varchar)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>date</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>DATE</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>date</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>TIMESTAMP(6)</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>datetimemin</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>TIMESTAMP(6)</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>decimal</td>
<td>32</td>
<td>38</td>
<td>0</td>
<td>38</td>
<td>-</td>
<td>-</td>
<td>DOUBLE</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>decimal</td>
<td>32</td>
<td>38</td>
<td>0</td>
<td>38</td>
<td>-</td>
<td>-</td>
<td>DECIMAL</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>decimaln</td>
<td>32</td>
<td>38</td>
<td>0</td>
<td>38</td>
<td>-</td>
<td>-</td>
<td>DECIMAL</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>float</td>
<td>-</td>
<td>4</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>REAL</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>float</td>
<td>-</td>
<td>8</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>DOUBLE</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>floatn</td>
<td>-</td>
<td>4</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>REAL</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>floatn</td>
<td>-</td>
<td>8</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>DOUBLE</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>image</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>BLOB</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>int</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>INTEGER</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>intn</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>INTEGER</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>money</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>DECIMAL</td>
<td>19</td>
<td>4</td>
<td>-</td>
</tr>
<tr>
<td>moneyn</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>DECIMAL</td>
<td>19</td>
<td>4</td>
<td>-</td>
</tr>
<tr>
<td>nchar</td>
<td>1</td>
<td>254</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>CHAR</td>
<td>-</td>
<td>-</td>
<td>N</td>
</tr>
<tr>
<td>nchar</td>
<td>255</td>
<td>32672</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>VARCHAR</td>
<td>-</td>
<td>-</td>
<td>N</td>
</tr>
<tr>
<td>nchar null (see nvarchar)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>numeric</td>
<td>1</td>
<td>31</td>
<td>0</td>
<td>31</td>
<td>-</td>
<td>-</td>
<td>DECIMAL</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>numeric</td>
<td>32</td>
<td>38</td>
<td>0</td>
<td>38</td>
<td>-</td>
<td>-</td>
<td>DOUBLE</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>numericn</td>
<td>1</td>
<td>31</td>
<td>0</td>
<td>31</td>
<td>-</td>
<td>-</td>
<td>DECIMAL</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>numericn</td>
<td>32</td>
<td>38</td>
<td>0</td>
<td>38</td>
<td>-</td>
<td>-</td>
<td>DOUBLE</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>nvarchar</td>
<td>1</td>
<td>32672</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>VARCHAR</td>
<td>-</td>
<td>-</td>
<td>N</td>
</tr>
<tr>
<td>real</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>REAL</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>smalldatetime</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>TIMESTAMP(6)</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>smallint</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>SMALLINT</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>smallmoney</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>DECIMAL</td>
<td>10</td>
<td>4</td>
<td>-</td>
</tr>
<tr>
<td>sysname</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>VARCHAR</td>
<td>30</td>
<td>-</td>
<td>N</td>
</tr>
<tr>
<td>text</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>CLOB</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>time</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>TIME</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>timestamp</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>VARCHAR</td>
<td>8</td>
<td>-</td>
<td>Y</td>
</tr>
</tbody>
</table>
### Default forward data type mappings for Sybase data sources

#### Table 227. Sybase CTLIB default forward data type mappings (continued)

<table>
<thead>
<tr>
<th>Remote Typename</th>
<th>Remote Lower Len</th>
<th>Remote Upper Len</th>
<th>Remote Lower Scale</th>
<th>Remote Upper Scale</th>
<th>Remote Bit Data</th>
<th>Remote Data Operators</th>
<th>Federated Typename</th>
<th>Federated Length</th>
<th>Federated Scale</th>
<th>Federated Bit Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>tinyint</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>SMALLINT</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>unichar²</td>
<td>1</td>
<td>254</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>CHAR</td>
<td>-</td>
<td>-</td>
<td>N</td>
</tr>
<tr>
<td>unichar²</td>
<td>255</td>
<td>32672</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>VARCHAR</td>
<td>-</td>
<td>-</td>
<td>N</td>
</tr>
</tbody>
</table>

unichar null (see univarchar)

| univarchar²     | 1                | 32672            | -                  | -                 | -               | -                     | VARCHAR            | -               | -              | N                 |
| varbinary       | 1                | 32672            | -                  | -                 | -               | -                     | VARCHAR            | -               | -              | Y                 |
| varchar         | 1                | 32672            | -                  | -                 | -               | -                     | VARCHAR            | -               | -              | N                 |

**Note:**

1. The federated type is TIMESTAMP(0) if the date_compat configuration parameter is set to ON.
2. Valid for non-Unicode federated databases.
Default forward data type mappings for Teradata data sources

The following table lists the default forward data type mappings for Teradata data sources.

Table 228. Teradata default forward data type mappings (Not all columns shown)

<table>
<thead>
<tr>
<th>Remote Tyename</th>
<th>Remote Lower Len</th>
<th>Remote Lower Scale</th>
<th>Remote Upper Len</th>
<th>Remote Upper Scale</th>
<th>Remote Bit Data</th>
<th>Remote Data Operators</th>
<th>Federated Tyename</th>
<th>Federated Length</th>
<th>Federated Scale</th>
<th>Federated Bit Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLOB</td>
<td>1</td>
<td>2097088000</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>BLOB</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>BYTE</td>
<td>1</td>
<td>254</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>CHAR</td>
<td>-</td>
<td>-</td>
<td>Y</td>
</tr>
<tr>
<td>BYTE</td>
<td>255</td>
<td>32672</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>VARCHAR</td>
<td>-</td>
<td>-</td>
<td>Y</td>
</tr>
<tr>
<td>BYTE</td>
<td>32673</td>
<td>64000</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>BLOB</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>BYTEINT</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>SMALLINT</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>CHAR</td>
<td>1</td>
<td>254</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>CHARACTER</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>CHAR</td>
<td>255</td>
<td>32672</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>VARCHAR</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>CHAR</td>
<td>32673</td>
<td>64000</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>CLOB</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>CLOB</td>
<td>1</td>
<td>2097088000</td>
<td>(Latin)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>CLOB</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>CLOB</td>
<td>1</td>
<td>1048544000</td>
<td>(Unicode)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>CLOB</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DATE</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>DATE</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DATE</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>TIMESTAMP</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>1</td>
<td>18</td>
<td>0</td>
<td>18</td>
<td>-</td>
<td>-</td>
<td>DECIMAL</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>DOUBLE</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>FLOAT</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>DOUBLE</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>GRAPHIC</td>
<td>1</td>
<td>127</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>GRAPHIC</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>GRAPHIC</td>
<td>128</td>
<td>16336</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>VARGRAPHIC</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>GRAPHIC</td>
<td>16337</td>
<td>32000</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>DBCLOB</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>INTEGER</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>INTEGER</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>INTERVAL</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>CHAR</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>NUMERIC</td>
<td>1</td>
<td>18</td>
<td>0</td>
<td>18</td>
<td>-</td>
<td>-</td>
<td>DECIMAL</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>REAL</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>DOUBLE</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>SMALLINT</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>TIME</td>
<td>0</td>
<td>21</td>
<td>0</td>
<td>21</td>
<td>-</td>
<td>-</td>
<td>TIME</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>TIMESTAMP(p)</td>
<td>-</td>
<td>-</td>
<td>p</td>
<td>p</td>
<td>-</td>
<td>-</td>
<td>TIMESTAMP(p)</td>
<td>10</td>
<td>6</td>
<td>-</td>
</tr>
<tr>
<td>VARBYTE</td>
<td>1</td>
<td>32762</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>VARCHAR</td>
<td>-</td>
<td>-</td>
<td>Y</td>
</tr>
<tr>
<td>VARBYTE</td>
<td>32763</td>
<td>64000</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>BLOB</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>1</td>
<td>32672</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>VARCHAR</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>32673</td>
<td>64000</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>CLOB</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>VARGRAPHIC</td>
<td>1</td>
<td>16336</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>VARGRAPHIC</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>VARGRAPHIC</td>
<td>16337</td>
<td>32000</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>DBCLOB</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Note:
1. The federated type is TIMESTAMP(0) if the date_compat configuration parameter is set to ON.
Default reverse data type mappings

For most data sources, the default type mappings are in the wrappers.

The two kinds of mappings between data source data types and federated
database data types are forward type mappings and reverse type mappings. In a
forward type mapping, the mapping is from a remote type to a comparable local
type. The other type of mapping is a reverse type mapping, which is used with
transparent DDL to create or modify remote tables.

The default type mappings for DB2 family data sources are in the DRDA wrapper.
The default type mappings for Informix are in the INFORMIX wrapper, and so
forth.

When you define a remote table or view to the federated database, the definition
includes a reverse type mapping. The mapping is from a local federated database
data type for each column, and the corresponding remote data type. For example,
there is a default reverse type mapping in which the local type REAL points to the
Informix type SMALLFLOAT.

Federated databases do not support mappings for LONG VARCHAR, LONG
VARGRAPHIC, and user-defined types.

When you use the CREATE TABLE statement to create a remote table, you specify
the local data types you want to include in the remote table. These default reverse
type mappings will assign corresponding remote types to these columns. For
example, suppose that you use the CREATE TABLE statement to define an
Informix table with a column C2. You specify BIGINT as the data type for C2 in
the statement. The default reverse type mapping of BIGINT depends on which
version of Informix you are creating the table on. The mapping for C2 in the
Informix table will be to DECIMAL in Informix Version 8 and to INT8 in Informix
Version 9.

You can override a default reverse type mapping, or create a new reverse type
mapping with the CREATE TYPE MAPPING statement.

The following tables show the default reverse mappings between federated
database local data types and remote data source data types.

These mappings are valid with all the supported versions, unless otherwise noted.
Default reverse data type mappings for DB2 Database for Linux, UNIX, and Windows data sources

The following table lists the default reverse data type mappings for DB2 Database for Linux, UNIX, and Windows data sources.

<table>
<thead>
<tr>
<th>Federated Typename</th>
<th>Federated Lower Len</th>
<th>Federated Upper Len</th>
<th>Federated Lower Scale</th>
<th>Federated Upper Scale</th>
<th>Federated Bit Data Operators</th>
<th>Remote Typename</th>
<th>Remote Length</th>
<th>Remote Scale</th>
<th>Remote Bit Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIGINT</td>
<td>-</td>
<td>8</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>BIGINT</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>BLOB</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>BLOB</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>CHARACTER</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>CHAR</td>
<td>-</td>
<td>-</td>
<td>Y</td>
</tr>
<tr>
<td>CHARACTER</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Y</td>
<td>-</td>
<td>CHAR</td>
<td>-</td>
<td>-</td>
<td>Y</td>
</tr>
<tr>
<td>CLOB</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>CLOB</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DATE</td>
<td>4</td>
<td>4</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>DATE</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DBCLOB</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>DBCLOB</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>DECIMAL</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DECFLOAT 2</td>
<td>8</td>
<td>8</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>DECFLOAT</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DECFLOAT 2</td>
<td>16</td>
<td>16</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>DECFLOAT</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>8</td>
<td>8</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>DOUBLE</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>FLOAT</td>
<td>8</td>
<td>8</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>DOUBLE</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>GRAPHIC</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>GRAPHIC</td>
<td>-</td>
<td>-</td>
<td>N</td>
</tr>
<tr>
<td>INTEGER</td>
<td>4</td>
<td>4</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>INTEGER</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>REAL</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>REAL</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>2</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>SMALLINT</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>TIME</td>
<td>3</td>
<td>3</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>TIME</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>TIMESTAMP(p)</td>
<td>-</td>
<td>p</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>TIMESTAMP(p)</td>
<td>p</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>VARCHAR</td>
<td>-</td>
<td>-</td>
<td>N</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Y</td>
<td>-</td>
<td>VARCHAR</td>
<td>-</td>
<td>-</td>
<td>Y</td>
</tr>
<tr>
<td>VARGRAPHIC</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>VARGRAPHIC</td>
<td>-</td>
<td>-</td>
<td>N</td>
</tr>
</tbody>
</table>

Note:
1. When the date_compatible parameter is set to OFF, the federated DATE maps to TIMESTAMP(0).
2. The SAME_DECFLT_ROUNDING server option is set to N by default and operations will not be pushed down to the remote data source unless SAME_DECFLT_ROUNDING is set to Y. For information on the SAME_DECFLT_ROUNDING server option, see DB2 database options reference.
3. For Version 9.5 or earlier, the remote scale for TIMESTAMP is 6.
Default reverse data type mappings for DB2 for System i data sources

The following table lists the default reverse data type mappings for DB2 for System i data sources.

<table>
<thead>
<tr>
<th>Federated Typename</th>
<th>Federated Lower Len</th>
<th>Federated Upper Len</th>
<th>Federated Lower Scale</th>
<th>Federated Upper Scale</th>
<th>Federated Bit Data</th>
<th>Remote Typename</th>
<th>Remote Length</th>
<th>Remote Scale</th>
<th>Remote Bit Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLOB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>BLOB</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>CHARACTER</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>CHARACTER</td>
<td>-</td>
<td>-</td>
<td>N</td>
</tr>
<tr>
<td>CHARACTER</td>
<td></td>
<td></td>
<td>-</td>
<td>Y</td>
<td></td>
<td>CHARACTER</td>
<td>-</td>
<td>-</td>
<td>Y</td>
</tr>
<tr>
<td>CLOB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>CLOB</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DATE</td>
<td>-</td>
<td>4</td>
<td>-</td>
<td>-</td>
<td></td>
<td>DATE</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DBCLOB</td>
<td>-</td>
<td></td>
<td>-</td>
<td>-</td>
<td></td>
<td>DBCLOB</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td>NUMERIC</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td>DECIMAL</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>-</td>
<td>8</td>
<td>-</td>
<td>-</td>
<td></td>
<td>FLOAT</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>GRAPHIC</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td>GRAPHIC</td>
<td>-</td>
<td>-</td>
<td>N</td>
</tr>
<tr>
<td>INTEGER</td>
<td>-</td>
<td>4</td>
<td>-</td>
<td>-</td>
<td></td>
<td>INTEGER</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>REAL</td>
<td>-</td>
<td>4</td>
<td>-</td>
<td>-</td>
<td></td>
<td>FLOAT</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>-</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td></td>
<td>SMALLINT</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>TIME</td>
<td>-</td>
<td>3</td>
<td>-</td>
<td>-</td>
<td></td>
<td>TIME</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>TIMESTAMP(p)</td>
<td>-</td>
<td>p</td>
<td>p</td>
<td>-</td>
<td></td>
<td>TIMESTAMP</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td>VARCHAR</td>
<td>-</td>
<td>-</td>
<td>N</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Y</td>
<td></td>
<td>VARCHAR</td>
<td>-</td>
<td>-</td>
<td>Y</td>
</tr>
<tr>
<td>VARGRAPHIC</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td>VARGRAPHIC</td>
<td>-</td>
<td>-</td>
<td>N</td>
</tr>
</tbody>
</table>
Default reverse data type mappings for DB2 for VM and VSE data sources

The following table lists the default reverse data type mappings for DB2 for VM and VSE data sources.

<table>
<thead>
<tr>
<th>Federated Typename</th>
<th>Federated Lower Len</th>
<th>Federated Upper Len</th>
<th>Federated Lower Scale</th>
<th>Federated Upper Scale</th>
<th>Bit Data</th>
<th>Operators</th>
<th>Remote Typename</th>
<th>Remote Length</th>
<th>Remote Scale</th>
<th>Remote Bit Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLOB</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>BLOB</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>CHARACTER</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>CHAR</td>
<td>-</td>
<td>-</td>
<td>Y</td>
</tr>
<tr>
<td>CHARACTER</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Y</td>
<td>-</td>
<td>-</td>
<td>CHAR</td>
<td>-</td>
<td>-</td>
<td>Y</td>
</tr>
<tr>
<td>CLOB</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>CLOB</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DATE</td>
<td>-</td>
<td>4</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>DATE</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DBCLOB</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>DBCLOB</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>DECIMAL</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>FLOAT</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>GRAPHIC</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>GRAPHIC</td>
<td>-</td>
<td>-</td>
<td>N</td>
</tr>
<tr>
<td>INTEGER</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>INTEGER</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>REAL</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>REAL</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>-</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>SMALLINT</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>TIME</td>
<td>-</td>
<td>3</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>TIME</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>TIMESTAMP(p)</td>
<td>-</td>
<td>p</td>
<td>p</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>TIMESTAMP</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>VARCHAR</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Y</td>
<td>-</td>
<td>-</td>
<td>VARCHAR</td>
<td>-</td>
<td>-</td>
<td>Y</td>
</tr>
<tr>
<td>VARGraph</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>VARGraph</td>
<td>-</td>
<td>-</td>
<td>N</td>
</tr>
</tbody>
</table>

Appendix E. Federated systems  995
Default reverse data type mappings for DB2 for z/OS data sources

The following table lists the default reverse data type mappings for DB2 for z/OS data sources.

<table>
<thead>
<tr>
<th>Federated Typename</th>
<th>Federated Lower Len</th>
<th>Federated Upper Len</th>
<th>Federated Lower Scale</th>
<th>Federated Upper Scale</th>
<th>Federated Bit Data</th>
<th>Remote Typename</th>
<th>Remote Length</th>
<th>Remote Scale</th>
<th>Remote Bit Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLOB</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>BLOB</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>CHARACTER</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>CHAR</td>
<td>-</td>
<td>-</td>
<td>N</td>
</tr>
<tr>
<td>CHARACTER</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Y</td>
<td>-</td>
<td>CHAR</td>
<td>-</td>
<td>-</td>
<td>Y</td>
</tr>
<tr>
<td>CLOB</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>CLOB</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DATE</td>
<td>-</td>
<td>4</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>DATE</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DBCLOB</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>DBCLOB</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>DECIMAL</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>-</td>
<td>8</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>DOUBLE</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>FLOAT</td>
<td>-</td>
<td>8</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>DOUBLE</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>GRAPHIC</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>GRAPHIC</td>
<td>-</td>
<td>-</td>
<td>N</td>
</tr>
<tr>
<td>INTEGER</td>
<td>-</td>
<td>4</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>INTEGER</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>REAL</td>
<td>-</td>
<td>4</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>REAL</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>-</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>SMALLINT</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>TIME</td>
<td>-</td>
<td>3</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>TIME</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>TIMESTAMP(p)</td>
<td>-</td>
<td>p</td>
<td>p</td>
<td>-</td>
<td>-</td>
<td>TIMESTAMP</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>VARCHAR</td>
<td>-</td>
<td>-</td>
<td>N</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Y</td>
<td>-</td>
<td>VARCHAR</td>
<td>-</td>
<td>-</td>
<td>Y</td>
</tr>
<tr>
<td>VARGRAPHIC</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>VARGRAPHIC</td>
<td>-</td>
<td>-</td>
<td>N</td>
</tr>
</tbody>
</table>
# Default reverse data type mappings for Informix data sources

The following table lists the default reverse data type mappings for Informix data sources.

<table>
<thead>
<tr>
<th>Federated Typename</th>
<th>Federated Lower Len</th>
<th>Federated Upper Len</th>
<th>Federated Lower Scale</th>
<th>Federated Upper Scale</th>
<th>Bit Data</th>
<th>Data Operators</th>
<th>Remote Typename</th>
<th>Remote Length</th>
<th>Remote Scale</th>
<th>Remote Bit Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIGINT ¹</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>DECIMAL</td>
<td>19</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>BIGINT ²</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>INT8</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>BLOB</td>
<td>1</td>
<td>2147483647</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>BYTE</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>CHARACTER</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>N</td>
<td>-</td>
<td>-</td>
<td>CHAR</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>CHARACTER</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Y</td>
<td>-</td>
<td>-</td>
<td>BYTE</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>CLOB</td>
<td>1</td>
<td>2147483647</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>TEXT</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DATE</td>
<td>-</td>
<td>4</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>DATE</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>DECIMAL</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>-</td>
<td>8</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>FLOAT</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>INTEGER</td>
<td>-</td>
<td>4</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>INTEGER</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>REAL</td>
<td>-</td>
<td>4</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>SMALLFLOAT</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>-</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>SMALLINT</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>TIME</td>
<td>-</td>
<td>3</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>DATETIME</td>
<td>6</td>
<td>10</td>
<td>-</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>-</td>
<td>10</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>DATETIME</td>
<td>0</td>
<td>15</td>
<td>-</td>
</tr>
<tr>
<td>VARCHAR 1</td>
<td>1</td>
<td>254</td>
<td>-</td>
<td>N</td>
<td>-</td>
<td>-</td>
<td>VARCHAR</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>VARCHAR ³</td>
<td>255</td>
<td>32672</td>
<td>-</td>
<td>N</td>
<td>-</td>
<td>-</td>
<td>TEXT</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Y</td>
<td>-</td>
<td>-</td>
<td>BYTE</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>VARCHAR ²</td>
<td>255</td>
<td>2048</td>
<td>-</td>
<td>N</td>
<td>-</td>
<td>-</td>
<td>LVARCHAR</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>VARCHAR ²</td>
<td>2049</td>
<td>32672</td>
<td>-</td>
<td>N</td>
<td>-</td>
<td>-</td>
<td>TEXT</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

**Note:**
1. This type mapping is valid only with Informix server Version 8 (or lower).
2. This type mapping is valid only with Informix server Version 9 (or higher).

For the Informix DATETIME data type, the federated server uses the Informix high-level qualifier as the REMOTE_LENGTH and the Informix low-level qualifier as the REMOTE_SCALE.

The Informix qualifiers are the "TU_" constants defined in the Informix Client SDK datatime.h file. The constants are:

<table>
<thead>
<tr>
<th>Qualifier</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>YEAR</td>
</tr>
<tr>
<td>2</td>
<td>MONTH</td>
</tr>
<tr>
<td>4</td>
<td>DAY</td>
</tr>
<tr>
<td>6</td>
<td>HOUR</td>
</tr>
<tr>
<td>8</td>
<td>MINUTE</td>
</tr>
<tr>
<td>10</td>
<td>SECOND</td>
</tr>
<tr>
<td>11</td>
<td>FRACTION(1)</td>
</tr>
<tr>
<td>12</td>
<td>FRACTION(2)</td>
</tr>
<tr>
<td>13</td>
<td>FRACTION(3)</td>
</tr>
<tr>
<td>14</td>
<td>FRACTION(4)</td>
</tr>
<tr>
<td>15</td>
<td>FRACTION(5)</td>
</tr>
</tbody>
</table>
Default reverse data type mappings for Microsoft SQL Server data sources

The following table lists the default reverse data type mappings for Microsoft SQL Server data sources.

<table>
<thead>
<tr>
<th>Federated Typename</th>
<th>Federated Lower Len</th>
<th>Federated Upper Len</th>
<th>Federated Lower Scale</th>
<th>Federated Upper Scale</th>
<th>Federated Bit Data</th>
<th>Federated Data Operators</th>
<th>Remote Typename</th>
<th>Remote Length</th>
<th>Remote Scale</th>
<th>Remote Bit Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIGINT</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>bigint</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>BLOB</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>image</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>CHARACTER</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Y</td>
<td>-</td>
<td>binary</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>CHARACTER</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>N</td>
<td>-</td>
<td>char</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>CLOB</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>text</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DATE</td>
<td>-</td>
<td>4</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>datetime</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>decimal</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>-</td>
<td>8</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>float</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>INTEGER</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>int</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>smallint</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>REAL</td>
<td>-</td>
<td>4</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>real</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>TIME</td>
<td>-</td>
<td>3</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>datetime</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>-</td>
<td>10</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>datetime</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>1</td>
<td>8000</td>
<td>-</td>
<td>-</td>
<td>N</td>
<td>-</td>
<td>varchar</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>8001</td>
<td>32672</td>
<td>-</td>
<td>-</td>
<td>N</td>
<td>-</td>
<td>text</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>1</td>
<td>8000</td>
<td>-</td>
<td>-</td>
<td>Y</td>
<td>-</td>
<td>varbinary</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>8001</td>
<td>32672</td>
<td>-</td>
<td>-</td>
<td>Y</td>
<td>-</td>
<td>image</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Note:
1. This type mapping is valid only with Microsoft SQL Server Version 2000.
Default reverse data type mappings for Oracle NET8 data sources

The following table lists the default reverse data type mappings for Oracle NET8 data sources.

Table 235. Oracle NET8 default reverse data type mappings

<table>
<thead>
<tr>
<th>Federated Typename</th>
<th>Federated Lower Len</th>
<th>Federated Upper Len</th>
<th>Federated Lower Scale</th>
<th>Federated Upper Scale</th>
<th>Federated Bit Data</th>
<th>Remote Typename</th>
<th>Remote Length</th>
<th>Remote Scale</th>
<th>Remote Bit Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIGINT</td>
<td>0</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>N</td>
<td>NUMBER</td>
<td>19</td>
<td>0</td>
<td>N</td>
</tr>
<tr>
<td>BLOB</td>
<td>0</td>
<td>2147483647</td>
<td>0</td>
<td>0</td>
<td>Y</td>
<td>BLOB</td>
<td>0</td>
<td>0</td>
<td>Y</td>
</tr>
<tr>
<td>CHARACTER</td>
<td>1</td>
<td>254</td>
<td>0</td>
<td>0</td>
<td>N</td>
<td>CHAR</td>
<td>0</td>
<td>0</td>
<td>N</td>
</tr>
<tr>
<td>CHARACTEN</td>
<td>1</td>
<td>254</td>
<td>0</td>
<td>0</td>
<td>Y</td>
<td>RAW</td>
<td>0</td>
<td>0</td>
<td>Y</td>
</tr>
<tr>
<td>CLOB</td>
<td>0</td>
<td>2147483647</td>
<td>0</td>
<td>0</td>
<td>N</td>
<td>CLOB</td>
<td>0</td>
<td>0</td>
<td>N</td>
</tr>
<tr>
<td>DATE</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>N</td>
<td>DATE</td>
<td>0</td>
<td>0</td>
<td>N</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>N</td>
<td>NUMBER</td>
<td>0</td>
<td>0</td>
<td>N</td>
</tr>
<tr>
<td>DECFLOAT</td>
<td>0</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>N</td>
<td>NUMBER</td>
<td>0</td>
<td>0</td>
<td>N</td>
</tr>
<tr>
<td>DECFLOAT</td>
<td>0</td>
<td>16</td>
<td>0</td>
<td>0</td>
<td>N</td>
<td>NUMBER</td>
<td>0</td>
<td>0</td>
<td>N</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>0</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>N</td>
<td>FLOAT</td>
<td>126</td>
<td>0</td>
<td>N</td>
</tr>
<tr>
<td>FLOAT</td>
<td>0</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>N</td>
<td>FLOAT</td>
<td>126</td>
<td>0</td>
<td>N</td>
</tr>
<tr>
<td>INTEGER</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>N</td>
<td>NUMBER</td>
<td>10</td>
<td>0</td>
<td>N</td>
</tr>
<tr>
<td>REAL</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>N</td>
<td>FLOAT</td>
<td>63</td>
<td>0</td>
<td>N</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>N</td>
<td>NUMBER</td>
<td>5</td>
<td>0</td>
<td>N</td>
</tr>
<tr>
<td>TIME</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>N</td>
<td>DATE</td>
<td>0</td>
<td>0</td>
<td>N</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>0</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>N</td>
<td>DATE</td>
<td>0</td>
<td>0</td>
<td>N</td>
</tr>
<tr>
<td>TIMESTAMP(p)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>N</td>
<td>TIMESTAMP(p)</td>
<td>-</td>
<td>-</td>
<td>N</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>1</td>
<td>4000</td>
<td>0</td>
<td>0</td>
<td>N</td>
<td>VARCHAR2</td>
<td>0</td>
<td>0</td>
<td>N</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>1</td>
<td>2000</td>
<td>0</td>
<td>0</td>
<td>Y</td>
<td>RAW</td>
<td>0</td>
<td>0</td>
<td>Y</td>
</tr>
</tbody>
</table>

Note:
1. When the date_compatible parameter is set to OFF, the federated DATE maps to the Oracle date. When the date_compatible parameter is set to ON, the federated DATE (equivalent to TIMESTAMP(0)), maps to Oracle TIMESTAMP(0).
2. This type mapping is valid only with Oracle Version 8.
3. TIMESTAMP(p) represents a timestamp with a variable scale from 0-9 for Oracle and 0-12 for federation. When the scale is 0-9, the remote Oracle TIMESTAMP has the same scale as the federated TIMESTAMP. If the scale of the federated TIMESTAMP is greater than 9, the corresponding scale of the Oracle TIMESTAMP is 9 which is the largest Oracle scale.
4. This type mapping is valid only with Oracle Version 9, 10, and 11.
## Default reverse data type mappings for Sybase data sources

The following table lists the default reverse data type mappings for Sybase data sources.

### Table 236. Sybase CTLIB default reverse data type mappings

<table>
<thead>
<tr>
<th>Federated Typename</th>
<th>Federated Lower Len</th>
<th>Federated Upper Len</th>
<th>Federated Lower Scale</th>
<th>Federated Upper Scale</th>
<th>Federated Bit Data</th>
<th>Federated Data Operators</th>
<th>Remote Typename</th>
<th>Remote Length</th>
<th>Remote Scale</th>
<th>Remote Bit Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIGINT</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>decimal</td>
<td>19</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>BLOB</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>image</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>CHARACTER</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>N</td>
<td>-</td>
<td>-</td>
<td>char</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>CHARACTER</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Y</td>
<td>-</td>
<td>-</td>
<td>binary</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>CLOB</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>text</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DATE</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>datetime</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>decimal</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>float</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>INTEGER</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>integer</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>REAL</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>real</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>smallint</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>TIME</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>datetime</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>datetime</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>VARCHAR&lt;sup&gt;1&lt;/sup&gt;</td>
<td>1</td>
<td>255</td>
<td>-</td>
<td>N</td>
<td>-</td>
<td>-</td>
<td>varchar</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>VARCHAR&lt;sup&gt;1&lt;/sup&gt;</td>
<td>256</td>
<td>32672</td>
<td>-</td>
<td>N</td>
<td>-</td>
<td>-</td>
<td>text</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>VARCHAR&lt;sup&gt;2&lt;/sup&gt;</td>
<td>1</td>
<td>16384</td>
<td>-</td>
<td>N</td>
<td>-</td>
<td>-</td>
<td>varchar</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>VARCHAR&lt;sup&gt;2&lt;/sup&gt;</td>
<td>16385</td>
<td>32672</td>
<td>-</td>
<td>N</td>
<td>-</td>
<td>-</td>
<td>text</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>VARCHAR&lt;sup&gt;2&lt;/sup&gt;</td>
<td>1</td>
<td>255</td>
<td>-</td>
<td>Y</td>
<td>-</td>
<td>-</td>
<td>varbinary</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>VARCHAR&lt;sup&gt;2&lt;/sup&gt;</td>
<td>256</td>
<td>32672</td>
<td>-</td>
<td>Y</td>
<td>-</td>
<td>-</td>
<td>image</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>VARCHAR&lt;sup&gt;2&lt;/sup&gt;</td>
<td>1</td>
<td>16384</td>
<td>-</td>
<td>Y</td>
<td>-</td>
<td>-</td>
<td>varbinary</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>VARCHAR&lt;sup&gt;2&lt;/sup&gt;</td>
<td>16385</td>
<td>32672</td>
<td>-</td>
<td>Y</td>
<td>-</td>
<td>-</td>
<td>image</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

### Note:

1. This type mapping is valid only for CTLIB with Sybase server version 12.0 (or earlier).
2. This type mapping is valid only for CTLIB with Sybase server version 12.5 (or later).
## Default reverse data type mappings for Teradata data sources

The following table lists the default reverse data type mappings for Teradata data sources.

<table>
<thead>
<tr>
<th>Federated Typename</th>
<th>Federated Lower Len</th>
<th>Federated Upper Len</th>
<th>Federated Lower Scale</th>
<th>Federated Upper Scale</th>
<th>Federated Bit Data Operators</th>
<th>Remote Typename</th>
<th>Remote Length</th>
<th>Remote Scale</th>
<th>Remote Bit Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLOB</td>
<td>1</td>
<td>2097088000</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>BLOB</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>CHARACTER</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>CHARACTER</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>CHARACTER</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Y</td>
<td>-</td>
<td>BYTE</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>CLOB</td>
<td>1</td>
<td>2097088000</td>
<td>-</td>
<td>-</td>
<td>CLOB</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DATE</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>DATE</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DBCLOB</td>
<td>1</td>
<td>64000</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>VARGRAPHIC</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>1</td>
<td>18</td>
<td>0</td>
<td>18</td>
<td>-</td>
<td>DECIMAL</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>19</td>
<td>31</td>
<td>0</td>
<td>31</td>
<td>-</td>
<td>FLOAT</td>
<td>8</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>FLOOR</td>
<td>FLOAT</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>GRAPHIC</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>GRAPHIC</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>INTEGER</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>INTEGER</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>REAL</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>FLOOR</td>
<td>FLOAT</td>
<td>8</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>SMALLINT</td>
<td>SMALLINT</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>TIME</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>TIME</td>
<td>15</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>10</td>
<td>10</td>
<td>6</td>
<td>6</td>
<td>-</td>
<td>TIMESTAMP</td>
<td>26</td>
<td>6</td>
<td>-</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>VARCHAR</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Y</td>
<td>-</td>
<td>VARCHAR</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>VARGRAPHIC</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>VARGRAPHIC</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

**Note:**
1. The Teradata VARGRAPHIC data type can contain only the specified length (1 to 32000) of a DBCLOB data type.
Default reverse data type mappings for Teradata data sources
Appendix F. The SAMPLE database

Sample database can be used for various purposes like testing your applications, trying different features of DB2 and so on. Most of the sample application programs under DB2PATH/sqlib/samples use sample database for demonstrating various features of DB2 that makes it easy to understand the technology.

Once the sample database is created, you will notice:

- Organizational schema for non-XML data and
- Purchase order schema for XML data are created.

The data and database objects under these schemas are created using real time environment on a small scale.

Following is a description of each of the tables in the SAMPLE database. Initial data values for each table are given; a dash (-) indicates a NULL value.

**ACT table**

<table>
<thead>
<tr>
<th>Name:</th>
<th>ACTNO</th>
<th>ACTKWD</th>
<th>ACTDESC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type:</td>
<td>SMALLINT</td>
<td>CHAR(6)</td>
<td>VARCHAR(20)</td>
</tr>
<tr>
<td>Values:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>MANAGE</td>
<td>MANAGE/ADVISE</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>ECOST</td>
<td>ESTIMATE COST</td>
</tr>
<tr>
<td></td>
<td>30</td>
<td>DEFINE</td>
<td>DEFINE SPECS</td>
</tr>
<tr>
<td></td>
<td>40</td>
<td>LEADPR</td>
<td>LEAD PROGRAM/DESIGN</td>
</tr>
<tr>
<td></td>
<td>50</td>
<td>SPECS</td>
<td>WRITE SPECS</td>
</tr>
<tr>
<td></td>
<td>60</td>
<td>LOGIC</td>
<td>DESCRIBE LOGIC</td>
</tr>
<tr>
<td></td>
<td>70</td>
<td>CODE</td>
<td>CODE PROGRAMS</td>
</tr>
<tr>
<td></td>
<td>80</td>
<td>TEST</td>
<td>TEST PROGRAMS</td>
</tr>
<tr>
<td></td>
<td>90</td>
<td>ADMQS</td>
<td>ADM QUERY SYSTEM</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>TEACH</td>
<td>TEACH CLASSES</td>
</tr>
<tr>
<td></td>
<td>110</td>
<td>COURSE</td>
<td>DEVELOP COURSES</td>
</tr>
<tr>
<td></td>
<td>120</td>
<td>STAFF</td>
<td>PERS AND STAFFING</td>
</tr>
<tr>
<td></td>
<td>130</td>
<td>OPERAT</td>
<td>OPER COMPUTER SYS</td>
</tr>
<tr>
<td></td>
<td>140</td>
<td>MAINT</td>
<td>MAINT SOFTWARE SYS</td>
</tr>
<tr>
<td></td>
<td>150</td>
<td>ADMSYS</td>
<td>ADM OPERATING SYS</td>
</tr>
<tr>
<td></td>
<td>160</td>
<td>ADMDB</td>
<td>ADM DATA BASES</td>
</tr>
<tr>
<td></td>
<td>170</td>
<td>ADMDC</td>
<td>ADM DATA COMM</td>
</tr>
<tr>
<td></td>
<td>180</td>
<td>DOC</td>
<td>DOCUMENT</td>
</tr>
</tbody>
</table>

**ADEFUSER table**

<table>
<thead>
<tr>
<th>Name:</th>
<th>WORKDEPT</th>
<th>NO_OF_EMPLOYEES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type:</td>
<td>CHAR(3)</td>
<td>INTEGER</td>
</tr>
<tr>
<td>Values:</td>
<td>A00</td>
<td>5</td>
</tr>
</tbody>
</table>

© Copyright IBM Corp. 1993, 2009
### The SAMPLE database

Table: **CL_SCHED**

<table>
<thead>
<tr>
<th>Class Code</th>
<th>DAY</th>
<th>Starting Time</th>
<th>Ending Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>042:BF</td>
<td>4</td>
<td>12:10:00</td>
<td>14:00:00</td>
</tr>
<tr>
<td>553:MJA</td>
<td>1</td>
<td>10:30:00</td>
<td>11:00:00</td>
</tr>
<tr>
<td>543:CWM</td>
<td>3</td>
<td>09:10:00</td>
<td>10:30:00</td>
</tr>
<tr>
<td>778:RES</td>
<td>2</td>
<td>12:10:00</td>
<td>14:00:00</td>
</tr>
<tr>
<td>044:HD</td>
<td>3</td>
<td>17:12:30</td>
<td>18:00:00</td>
</tr>
</tbody>
</table>

Table: **DEPT**

<table>
<thead>
<tr>
<th>DEPTNO</th>
<th>DEPTNAME</th>
<th>MGRNO</th>
<th>ADMRDEPT</th>
<th>LOCATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>A00</td>
<td>SPIFFY COMPUTER SERVICE DIV.</td>
<td>000010</td>
<td>A00</td>
<td>A00</td>
</tr>
<tr>
<td>B01</td>
<td>PLANNING</td>
<td>000000</td>
<td>A00</td>
<td>A00</td>
</tr>
<tr>
<td>C01</td>
<td>INFORMATION CENTER</td>
<td>000050</td>
<td>A00</td>
<td>A00</td>
</tr>
<tr>
<td>D01</td>
<td>DEVELOPMENT CENTER</td>
<td>000060</td>
<td>D01</td>
<td>D01</td>
</tr>
<tr>
<td>D11</td>
<td>MANUFACTURING SYSTEMS</td>
<td>000070</td>
<td>D01</td>
<td>D01</td>
</tr>
<tr>
<td>D21</td>
<td>ADMINISTRATION SYSTEMS</td>
<td>000080</td>
<td>E01</td>
<td>E01</td>
</tr>
<tr>
<td>E01</td>
<td>SUPPORT SERVICES</td>
<td>000090</td>
<td>E01</td>
<td>E01</td>
</tr>
<tr>
<td>E11</td>
<td>OPERATIONS</td>
<td>000100</td>
<td>E01</td>
<td>E01</td>
</tr>
<tr>
<td>E21</td>
<td>SOFTWARE SUPPORT</td>
<td>000110</td>
<td>E01</td>
<td>E01</td>
</tr>
<tr>
<td>F22</td>
<td>BRANCH OFFICE F2</td>
<td>000120</td>
<td>E01</td>
<td>E01</td>
</tr>
<tr>
<td>G22</td>
<td>BRANCH OFFICE G2</td>
<td>000130</td>
<td>E01</td>
<td>E01</td>
</tr>
<tr>
<td>H22</td>
<td>BRANCH OFFICE H2</td>
<td>000140</td>
<td>E01</td>
<td>E01</td>
</tr>
<tr>
<td>I22</td>
<td>BRANCH OFFICE I2</td>
<td>000150</td>
<td>E01</td>
<td>E01</td>
</tr>
<tr>
<td>J22</td>
<td>BRANCH OFFICE J2</td>
<td>000160</td>
<td>E01</td>
<td>E01</td>
</tr>
</tbody>
</table>
### DEPARTMENT table

<table>
<thead>
<tr>
<th>Name</th>
<th>DEPTNO</th>
<th>DEPTNAME</th>
<th>MGRNO</th>
<th>ADMRDEPT</th>
<th>LOCATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>CHAR(3) Not Null</td>
<td>VARCHAR(29) Not Null</td>
<td>CHAR(6)</td>
<td>CHAR(3) Not Null</td>
<td>CHAR(16)</td>
</tr>
<tr>
<td>Desc</td>
<td>Department number</td>
<td>Name describing general activities of department</td>
<td>Employee number (EMPNO) of department manager</td>
<td>Department (DEPTNO) to which this department reports</td>
<td>Name of the remote location</td>
</tr>
<tr>
<td>Values</td>
<td>A00</td>
<td>SPIFFY COMPUTER SERVICE DIV</td>
<td>000010</td>
<td>A00</td>
<td></td>
</tr>
<tr>
<td></td>
<td>B01</td>
<td>PLANNING</td>
<td>000020</td>
<td>A00</td>
<td></td>
</tr>
<tr>
<td></td>
<td>C01</td>
<td>INFORMATION CENTER</td>
<td>000030</td>
<td>A00</td>
<td></td>
</tr>
<tr>
<td></td>
<td>D01</td>
<td>DEVELOPMENT CENTER</td>
<td></td>
<td>A00</td>
<td></td>
</tr>
<tr>
<td></td>
<td>D11</td>
<td>MANUFACTURING SYSTEMS</td>
<td>000060</td>
<td>D01</td>
<td></td>
</tr>
<tr>
<td></td>
<td>D21</td>
<td>ADMINISTRATION SYSTEMS</td>
<td>000070</td>
<td>D01</td>
<td></td>
</tr>
<tr>
<td></td>
<td>E01</td>
<td>SUPPORT SERVICES</td>
<td>000050</td>
<td>A00</td>
<td></td>
</tr>
<tr>
<td></td>
<td>E11</td>
<td>OPERATIONS</td>
<td>000090</td>
<td>E01</td>
<td></td>
</tr>
<tr>
<td></td>
<td>E21</td>
<td>SOFTWARE SUPPORT</td>
<td>000100</td>
<td>E01</td>
<td></td>
</tr>
<tr>
<td></td>
<td>F22</td>
<td>BRANCH OFFICE F2</td>
<td></td>
<td>E01</td>
<td></td>
</tr>
<tr>
<td></td>
<td>G22</td>
<td>BRANCH OFFICE G2</td>
<td></td>
<td>E01</td>
<td></td>
</tr>
<tr>
<td></td>
<td>H22</td>
<td>BRANCH OFFICE H2</td>
<td></td>
<td>E01</td>
<td></td>
</tr>
<tr>
<td></td>
<td>I22</td>
<td>BRANCH OFFICE I2</td>
<td></td>
<td>E01</td>
<td></td>
</tr>
<tr>
<td></td>
<td>J22</td>
<td>BRANCH OFFICE J2</td>
<td></td>
<td>E01</td>
<td></td>
</tr>
</tbody>
</table>

### EMPLOYEE and EMP tables

These two tables have identical content.

<table>
<thead>
<tr>
<th>Name</th>
<th>EMPNO</th>
<th>FIRSTNME</th>
<th>MIDINIT</th>
<th>LASTNAME</th>
<th>WORKDEPT</th>
<th>PHONENO</th>
<th>HIREDATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>CHAR(6) Not Null</td>
<td>VARCHAR(12) Not Null</td>
<td>CHAR(1) Not Null</td>
<td>VARCHAR(15) Not Null</td>
<td>CHAR(3)</td>
<td>CHAR(4)</td>
<td>DATE</td>
</tr>
<tr>
<td>Desc</td>
<td>Employee number</td>
<td>First name</td>
<td>Middle initial</td>
<td>Last name</td>
<td>Department (DEPTNO) in which the employee works</td>
<td>Phone number</td>
<td>Date of hire</td>
</tr>
</tbody>
</table>

The following table contains the values in the EMPLOYEE table.
The SAMPLE database
MID
INIT

WORK
DEPT

PHONE
NO

VARCHAR(15)
Not Null

CHAR(3)

CHAR(4)

DATE

CHAR(8)

SMALLINT CHAR(1)
Not Null

DATE

DECIMAL DECIMAL DECIMAL
(9,2)
(9,2)
(9,2)

I

HAAS

A00

3978

1965-01-01

PRES

18

F

1933-08-24

52750

1000

L

THOMPSON

B01

3476

1973-10-10

MANAGER

18

M

1948-02-02

41250

800

3300

SALLY

A

KWAN

C01

4738

1975-04-05

MANAGER

20

F

1941-05-11

38250

800

3060

000050

JOHN

B

GEYER

E01

6789

1949-08-17

MANAGER

16

M

1925-09-15

40175

800

3214

000060

IRVING

F

STERN

D11

6423

1973-09-14

MANAGER

16

M

1945-07-07

32250

500

2580

000070

EVA

D

PULASKI

D21

7831

1980-09-30

MANAGER

16

F

1953-05-26

36170

700

2893

000090

EILEEN

W

HENDERSON

E11

5498

1970-08-15

MANAGER

16

F

1941-05-15

29750

600

2380

000100

THEODORE

Q

SPENSER

E21

0972

1980-06-19

MANAGER

14

M

1956-12-18

26150

500

2092

000110

VINCENZO

G

LUCCHESSI

A00

3490

1958-05-16

SALESREP

19

M

1929-11-05

46500

900

3720

000120

SEAN

O’CONNELL

A00

2167

1963-12-05

CLERK

14

M

1942-10-18

29250

600

2340

000130

DOLORES

M

QUINTANA

C01

4578

1971-07-28

ANALYST

16

F

1925-09-15

23800

500

1904

000140

HEATHER

A

NICHOLLS

C01

1793

1976-12-15

ANALYST

18

F

1946-01-19

28420

600

2274

000150

BRUCE

000160

ELIZABETH

000170

MASATOSHI

J

YOSHIMURA

D11

2890

1978-09-15

DESIGNER

16

M

1951-01-05

24680

500

1974

000180

MARILYN

S

SCOUTTEN

D11

1682

1973-07-07

DESIGNER

17

F

1949-02-21

21340

500

1707

000190

JAMES

H

WALKER

D11

2986

1974-07-26

DESIGNER

16

M

1952-06-25

20450

400

1636

000200

DAVID

BROWN

D11

4501

1966-03-03

DESIGNER

16

M

1941-05-29

27740

600

2217

000210

WILLIAM

JONES

D11

0942

1979-04-11

DESIGNER

17

M

1953-02-23

18270

400

1462

000220

JENNIFER

K

LUTZ

D11

0672

1968-08-29

DESIGNER

18

F

1948-03-19

29840

600

2387

000230

JAMES

J

JEFFERSON

D21

2094

1966-11-21

CLERK

14

M

1935-05-30

22180

400

1774

000240

SALVATORE

M

MARINO

D21

3780

1979-12-05

CLERK

17

M

1954-03-31

28760

600

2301

000250

DANIEL

S

SMITH

D21

0961

1969-10-30

CLERK

15

M

1939-11-12

19180

400

1534

000260

SYBIL

P

JOHNSON

D21

8953

1975-09-11

CLERK

16

F

1936-10-05

17250

300

1380

000270

MARIA

L

PEREZ

D21

9001

1980-09-30

CLERK

15

F

1953-05-26

27380

500

2190

000280

ETHEL

R

SCHNEIDER

E11

8997

1967-03-24

OPERATOR

17

F

1936-03-28

26250

500

2100

000290

JOHN

R

PARKER

E11

4502

1980-05-30

OPERATOR

12

M

1946-07-09

15340

300

1227

000300

PHILIP

X

SMITH

E11

2095

1972-06-19

OPERATOR

14

M

1936-10-27

17750

400

1420

000310

MAUDE

F

SETRIGHT

E11

3332

1964-09-12

OPERATOR

12

F

1931-04-21

15900

300

1272

000320

RAMLAL

V

MEHTA

E21

9990

1965-07-07

FIELDREP

16

M

1932-08-11

19950

400

1596

000330

WING

LEE

E21

2103

1976-02-23

FIELDREP

14

M

1941-07-18

25370

500

2030

000340

JASON

GOUNOT

E21

5698

1947-05-05

FIELDREP

16

M

1926-05-17

23840

500

1907

EMPNO

FIRSTNME

CHAR(6)
Not Null

VARCHAR(12)
Not Null

CHAR(1)
Not Null

000010

CHRISTINE

000020

MICHAEL

000030

R

T

R

LASTNAME

HIREDATE

JOB

ED
LEVEL

SEX

BIRTHDATE

SALARY

BONUS

D11

4510

1972-02-12

DESIGNER

16

M

1947-05-17

25280

500

2022

PIANKA

D11

3782

1977-10-11

DESIGNER

17

F

1955-04-12

22250

400

1780

Name:

EMPNO

PROJNO

ACTNO

Type:

CHAR(6) Not Null

CHAR(6) Not Null

SMALLINT Not Null

Desc:

Employee number

Project number

Activity number

1006

4220

ADAMSON

EMP_ACT table

Values:

COMM

EMPTIME
DEC(5,2)
Proportion of employee’s
time spent on project

EMSTDATE

EMENDATE

DATE

DATE

Date activity starts

Date activity ends

000010

AD3100

10

.50

1982-01-01

1982-07-01

000070

AD3110

10

1.00

1982-01-01

1983-02-01

000230

AD3111

60

1.00

1982-01-01

1982-03-15

000230

AD3111

60

.50

1982-03-15

1982-04-15

000230

AD3111

70

.50

1982-03-15

1982-10-15

000230

AD3111

80

.50

1982-04-15

1982-10-15

000230

AD3111

180

1.00

1982-10-15

1983-01-01

000240

AD3111

70

1.00

1982-02-15

1982-09-15

000240

AD3111

80

1.00

1982-09-15

1983-01-01

000250

AD3112

60

1.00

1982-01-01

1982-02-01

000250

AD3112

60

.50

1982-02-01

1982-03-15

000250

AD3112

60

.50

1982-12-01

1983-01-01

000250

AD3112

60

1.00

1983-01-01

1983-02-01

000250

AD3112

70

.50

1982-02-01

1982-03-15

000250

AD3112

70

1.00

1982-03-15

1982-08-15

000250

AD3112

70

.25

1982-08-15

1982-10-15

000250

AD3112

80

.25

1982-08-15

1982-10-15

000250

AD3112

80

.50

1982-10-15

1982-12-01

000250

AD3112

180

.50

1982-08-15

1983-01-01

000260

AD3113

70

.50

1982-06-15

1982-07-01

000260

AD3113

70

1.00

1982-07-01

1983-02-01

000260

AD3113

80

1.00

1982-01-01

1982-03-01

000260

AD3113

80

.50

1982-03-01

1982-04-15

000260

AD3113

180

.50

1982-03-01

1982-04-15

000260

AD3113

180

1.00

1982-04-15

1982-06-01

000260

AD3113

180

.50

1982-06-01

1982-07-01

000270

AD3113

60

.50

1982-03-01

1982-04-01

000270

AD3113

60

1.00

1982-04-01

1982-09-01

000270

AD3113

60

.25

1982-09-01

1982-10-15

SQL Reference, Volume 1


### EMP_PHOTO table

<table>
<thead>
<tr>
<th>Name:</th>
<th>EMPNO</th>
<th>PHOTO_FORMAT</th>
<th>PICTURE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type:</td>
<td>CHAR(6) Not Null</td>
<td>VARCHAR(10) Not Null</td>
<td>BLOB(100K)</td>
</tr>
<tr>
<td>Desc:</td>
<td>Employee number</td>
<td>Photo format</td>
<td>Photo of employee</td>
</tr>
<tr>
<td>Values:</td>
<td>000130</td>
<td>bitmap</td>
<td>db200130.bmp</td>
</tr>
<tr>
<td></td>
<td>000140</td>
<td>bitmap</td>
<td>db200140.bmp</td>
</tr>
<tr>
<td></td>
<td>000150</td>
<td>bitmap</td>
<td>db200150.bmp</td>
</tr>
<tr>
<td></td>
<td>000190</td>
<td>bitmap</td>
<td>db200190.bmp</td>
</tr>
<tr>
<td></td>
<td>000200</td>
<td>bitmap</td>
<td>db200200.bmp</td>
</tr>
<tr>
<td>Name:</td>
<td>EMPNO</td>
<td>PROJNO</td>
<td>ACTNO</td>
</tr>
<tr>
<td>------</td>
<td>--------</td>
<td>---------</td>
<td>-------</td>
</tr>
<tr>
<td>Type:</td>
<td>CHAR(6)</td>
<td>CHAR(6)</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>Values:</td>
<td>000070</td>
<td>AD3110</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>000230</td>
<td>AD3111</td>
<td>60</td>
</tr>
<tr>
<td></td>
<td>000250</td>
<td>AD3111</td>
<td>60</td>
</tr>
<tr>
<td></td>
<td>000250</td>
<td>AD3111</td>
<td>70</td>
</tr>
<tr>
<td></td>
<td>000250</td>
<td>AD3111</td>
<td>80</td>
</tr>
<tr>
<td></td>
<td>000250</td>
<td>AD3111</td>
<td>180</td>
</tr>
<tr>
<td></td>
<td>000240</td>
<td>AD3111</td>
<td>70</td>
</tr>
<tr>
<td></td>
<td>000240</td>
<td>AD3111</td>
<td>80</td>
</tr>
<tr>
<td></td>
<td>000250</td>
<td>AD3112</td>
<td>60</td>
</tr>
<tr>
<td></td>
<td>000250</td>
<td>AD3112</td>
<td>60</td>
</tr>
<tr>
<td></td>
<td>000250</td>
<td>AD3112</td>
<td>70</td>
</tr>
<tr>
<td></td>
<td>000250</td>
<td>AD3112</td>
<td>70</td>
</tr>
<tr>
<td></td>
<td>000250</td>
<td>AD3112</td>
<td>80</td>
</tr>
<tr>
<td></td>
<td>000250</td>
<td>AD3112</td>
<td>80</td>
</tr>
<tr>
<td></td>
<td>000250</td>
<td>AD3112</td>
<td>180</td>
</tr>
<tr>
<td></td>
<td>000260</td>
<td>AD3113</td>
<td>70</td>
</tr>
<tr>
<td></td>
<td>000260</td>
<td>AD3113</td>
<td>70</td>
</tr>
<tr>
<td></td>
<td>000260</td>
<td>AD3113</td>
<td>80</td>
</tr>
<tr>
<td></td>
<td>000260</td>
<td>AD3113</td>
<td>80</td>
</tr>
<tr>
<td></td>
<td>000260</td>
<td>AD3113</td>
<td>180</td>
</tr>
<tr>
<td></td>
<td>000260</td>
<td>AD3113</td>
<td>180</td>
</tr>
<tr>
<td></td>
<td>000260</td>
<td>AD3113</td>
<td>180</td>
</tr>
<tr>
<td></td>
<td>000270</td>
<td>AD3113</td>
<td>60</td>
</tr>
<tr>
<td></td>
<td>000270</td>
<td>AD3113</td>
<td>60</td>
</tr>
<tr>
<td></td>
<td>000270</td>
<td>AD3113</td>
<td>60</td>
</tr>
<tr>
<td></td>
<td>000270</td>
<td>AD3113</td>
<td>70</td>
</tr>
<tr>
<td></td>
<td>000270</td>
<td>AD3113</td>
<td>70</td>
</tr>
<tr>
<td></td>
<td>000270</td>
<td>AD3113</td>
<td>80</td>
</tr>
<tr>
<td></td>
<td>000270</td>
<td>AD3113</td>
<td>80</td>
</tr>
<tr>
<td></td>
<td>000050</td>
<td>IF1000</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>000130</td>
<td>IF1000</td>
<td>90</td>
</tr>
<tr>
<td></td>
<td>000130</td>
<td>IF1000</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>000140</td>
<td>IF1000</td>
<td>90</td>
</tr>
<tr>
<td></td>
<td>000030</td>
<td>IF2000</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>000140</td>
<td>IF2000</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>000140</td>
<td>IF2000</td>
<td>100</td>
</tr>
</tbody>
</table>
The SAMPLE database

<table>
<thead>
<tr>
<th>Name:</th>
<th>EMPNO</th>
<th>PROJNO</th>
<th>ACTNO</th>
<th>EMPTIME</th>
<th>EMSTDATE</th>
<th>EMENDATE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>000140</td>
<td>IF2000</td>
<td>110</td>
<td>0.50</td>
<td>03/01/1982</td>
<td>07/01/1982</td>
</tr>
<tr>
<td></td>
<td>000140</td>
<td>IF2000</td>
<td>110</td>
<td>0.50</td>
<td>10/01/1982</td>
<td>01/01/1983</td>
</tr>
<tr>
<td></td>
<td>000010</td>
<td>MA2100</td>
<td>10</td>
<td>0.50</td>
<td>01/01/1982</td>
<td>11/01/1982</td>
</tr>
<tr>
<td></td>
<td>000110</td>
<td>MA2100</td>
<td>20</td>
<td>1.00</td>
<td>01/01/1982</td>
<td>03/01/1983</td>
</tr>
<tr>
<td></td>
<td>000010</td>
<td>MA2110</td>
<td>10</td>
<td>1.00</td>
<td>01/01/1982</td>
<td>02/01/1983</td>
</tr>
<tr>
<td></td>
<td>000200</td>
<td>MA2111</td>
<td>50</td>
<td>1.00</td>
<td>01/01/1982</td>
<td>06/15/1982</td>
</tr>
<tr>
<td></td>
<td>000200</td>
<td>MA2111</td>
<td>60</td>
<td>1.00</td>
<td>06/15/1982</td>
<td>02/01/1983</td>
</tr>
<tr>
<td></td>
<td>000220</td>
<td>MA2111</td>
<td>40</td>
<td>1.00</td>
<td>01/01/1982</td>
<td>02/01/1983</td>
</tr>
<tr>
<td></td>
<td>000150</td>
<td>MA2112</td>
<td>60</td>
<td>1.00</td>
<td>01/01/1982</td>
<td>07/15/1982</td>
</tr>
<tr>
<td></td>
<td>000150</td>
<td>MA2112</td>
<td>180</td>
<td>1.00</td>
<td>07/15/1982</td>
<td>02/01/1983</td>
</tr>
<tr>
<td></td>
<td>000170</td>
<td>MA2112</td>
<td>60</td>
<td>1.00</td>
<td>01/01/1982</td>
<td>06/01/1983</td>
</tr>
<tr>
<td></td>
<td>000170</td>
<td>MA2112</td>
<td>70</td>
<td>1.00</td>
<td>06/01/1982</td>
<td>02/01/1983</td>
</tr>
<tr>
<td></td>
<td>000190</td>
<td>MA2112</td>
<td>70</td>
<td>1.00</td>
<td>01/01/1982</td>
<td>10/01/1982</td>
</tr>
<tr>
<td></td>
<td>000190</td>
<td>MA2112</td>
<td>80</td>
<td>1.00</td>
<td>10/01/1982</td>
<td>10/01/1982</td>
</tr>
<tr>
<td></td>
<td>000160</td>
<td>MA2113</td>
<td>60</td>
<td>1.00</td>
<td>07/15/1982</td>
<td>02/01/1983</td>
</tr>
<tr>
<td></td>
<td>000170</td>
<td>MA2113</td>
<td>80</td>
<td>1.00</td>
<td>01/01/1982</td>
<td>02/01/1983</td>
</tr>
<tr>
<td></td>
<td>000180</td>
<td>MA2113</td>
<td>70</td>
<td>1.00</td>
<td>04/01/1982</td>
<td>06/15/1982</td>
</tr>
<tr>
<td></td>
<td>000210</td>
<td>MA2113</td>
<td>80</td>
<td>0.50</td>
<td>10/01/1982</td>
<td>02/01/1983</td>
</tr>
<tr>
<td></td>
<td>000210</td>
<td>MA2113</td>
<td>180</td>
<td>0.50</td>
<td>10/01/1982</td>
<td>02/01/1983</td>
</tr>
<tr>
<td></td>
<td>000050</td>
<td>OP1000</td>
<td>10</td>
<td>0.25</td>
<td>01/01/1982</td>
<td>02/01/1983</td>
</tr>
<tr>
<td></td>
<td>000090</td>
<td>OP1010</td>
<td>10</td>
<td>1.00</td>
<td>01/01/1982</td>
<td>02/01/1983</td>
</tr>
<tr>
<td></td>
<td>000280</td>
<td>OP1010</td>
<td>130</td>
<td>1.00</td>
<td>01/01/1982</td>
<td>02/01/1983</td>
</tr>
<tr>
<td></td>
<td>000290</td>
<td>OP1010</td>
<td>130</td>
<td>1.00</td>
<td>01/01/1982</td>
<td>02/01/1983</td>
</tr>
<tr>
<td></td>
<td>000300</td>
<td>OP1010</td>
<td>130</td>
<td>1.00</td>
<td>01/01/1982</td>
<td>02/01/1983</td>
</tr>
<tr>
<td></td>
<td>000310</td>
<td>OP1010</td>
<td>130</td>
<td>1.00</td>
<td>01/01/1982</td>
<td>02/01/1983</td>
</tr>
<tr>
<td></td>
<td>000050</td>
<td>OP1010</td>
<td>10</td>
<td>0.75</td>
<td>01/01/1982</td>
<td>02/01/1983</td>
</tr>
<tr>
<td></td>
<td>000100</td>
<td>OP1010</td>
<td>10</td>
<td>1.00</td>
<td>01/01/1982</td>
<td>02/01/1983</td>
</tr>
<tr>
<td></td>
<td>000320</td>
<td>OP2011</td>
<td>140</td>
<td>0.75</td>
<td>01/01/1982</td>
<td>02/01/1983</td>
</tr>
<tr>
<td></td>
<td>000320</td>
<td>OP2011</td>
<td>150</td>
<td>0.25</td>
<td>01/01/1982</td>
<td>02/01/1983</td>
</tr>
<tr>
<td></td>
<td>000330</td>
<td>OP2012</td>
<td>140</td>
<td>0.25</td>
<td>01/01/1982</td>
<td>02/01/1983</td>
</tr>
<tr>
<td></td>
<td>000330</td>
<td>OP2012</td>
<td>160</td>
<td>0.75</td>
<td>01/01/1982</td>
<td>02/01/1983</td>
</tr>
<tr>
<td></td>
<td>000340</td>
<td>OP2013</td>
<td>140</td>
<td>0.50</td>
<td>01/01/1982</td>
<td>02/01/1983</td>
</tr>
<tr>
<td></td>
<td>000340</td>
<td>OP2013</td>
<td>170</td>
<td>0.50</td>
<td>01/01/1982</td>
<td>02/01/1983</td>
</tr>
<tr>
<td></td>
<td>000020</td>
<td>PL2100</td>
<td>30</td>
<td>1.00</td>
<td>01/01/1982</td>
<td>09/15/1982</td>
</tr>
</tbody>
</table>

**EMP_RESUME table**

<table>
<thead>
<tr>
<th>Name:</th>
<th>EMPNO</th>
<th>RESUME_FORMAT</th>
<th>RESUME</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type:</td>
<td>CHAR(6) Not Null</td>
<td>VARCHAR(10) Not Null</td>
<td>CLOB(5K)</td>
</tr>
<tr>
<td>Desc:</td>
<td>Employee number</td>
<td>Resume Format</td>
<td>Resume of employee</td>
</tr>
<tr>
<td>Values:</td>
<td>000130</td>
<td>ascii</td>
<td>db200130.asc</td>
</tr>
</tbody>
</table>
The SAMPLE database

<table>
<thead>
<tr>
<th>Name</th>
<th>EMPNO</th>
<th>RESUME_FORMAT</th>
<th>RESUME</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>000130</td>
<td>html</td>
<td>db200130.htm</td>
</tr>
<tr>
<td></td>
<td>000140</td>
<td>ascii</td>
<td>db200140.asc</td>
</tr>
<tr>
<td></td>
<td>000140</td>
<td>html</td>
<td>db200140.htm</td>
</tr>
<tr>
<td></td>
<td>000150</td>
<td>ascii</td>
<td>db200150.asc</td>
</tr>
<tr>
<td></td>
<td>000150</td>
<td>html</td>
<td>db200150.htm</td>
</tr>
<tr>
<td></td>
<td>000190</td>
<td>ascii</td>
<td>db200190.asc</td>
</tr>
<tr>
<td></td>
<td>000190</td>
<td>html</td>
<td>db200190.htm</td>
</tr>
</tbody>
</table>

IN_TRAY table

<table>
<thead>
<tr>
<th>Name</th>
<th>RECEIVED</th>
<th>SOURCE</th>
<th>SUBJECT</th>
<th>NOTE_TEXT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Date and Time received</td>
<td>User id of person sending note</td>
<td>Brief description</td>
<td>The note</td>
</tr>
<tr>
<td></td>
<td>1988-12-25-17.12.30.000000</td>
<td>BADAMSON</td>
<td>FWD: Fantastic year! 4th Quarter Bonus.</td>
<td>To: JWALKER Cc: QUINTANA, NICHOLLS Jim, Looks like our hard work has paid off. I have some good beer in the fridge if you want to come over to celebrate a bit. Delores and Heather, are you interested as well? Bruce &lt;Forwarding from ISTERN&gt; Subject: FWD: Fantastic year! 4th Quarter Bonus. To: Dept_D11 Congratulations on a job well done. Enjoy this year’s bonus. Irv &lt;Forwarding from CHAAS&gt; Subject: Fantastic year! 4th Quarter Bonus. To: All_Managers Our 4th quarter results are in. We pulled together as a team and exceeded our plan! I am pleased to announce a bonus this year of 18%. Enjoy the holidays. Christine Haas</td>
</tr>
</tbody>
</table>
## The SAMPLE database

### ORG table

<table>
<thead>
<tr>
<th>Name:</th>
<th>DEPTNUMB</th>
<th>DEPTNAME</th>
<th>MANAGER</th>
<th>DIVISION</th>
<th>LOCATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type:</td>
<td>SMALLINT Not Null</td>
<td>VARCHAR(14)</td>
<td>SMALLINT</td>
<td>VARCHAR(10)</td>
<td>VARCHAR(13)</td>
</tr>
<tr>
<td>Desc:</td>
<td>Department number</td>
<td>Department name</td>
<td>Manager number</td>
<td>Division of corporation</td>
<td>City</td>
</tr>
<tr>
<td>Values:</td>
<td>10</td>
<td>Head Office</td>
<td>160</td>
<td>Corporate</td>
<td>New York</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>New England</td>
<td>50</td>
<td>Eastern</td>
<td>Boston</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>Mid Atlantic</td>
<td>10</td>
<td>Eastern</td>
<td>Washington</td>
</tr>
<tr>
<td></td>
<td>38</td>
<td>South Atlantic</td>
<td>30</td>
<td>Eastern</td>
<td>Atlanta</td>
</tr>
<tr>
<td></td>
<td>42</td>
<td>Great Lakes</td>
<td>100</td>
<td>Midwest</td>
<td>Chicago</td>
</tr>
<tr>
<td></td>
<td>51</td>
<td>Plains</td>
<td>140</td>
<td>Midwest</td>
<td>Dallas</td>
</tr>
<tr>
<td></td>
<td>66</td>
<td>Pacific</td>
<td>270</td>
<td>Western</td>
<td>San Francisco</td>
</tr>
<tr>
<td></td>
<td>84</td>
<td>Mountain</td>
<td>290</td>
<td>Western</td>
<td>Denver</td>
</tr>
</tbody>
</table>

### PROJ table

<table>
<thead>
<tr>
<th>Name:</th>
<th>PROJNO</th>
<th>PROJNAME</th>
<th>DEPTNO</th>
<th>RESPEMP</th>
<th>PRSTAFF</th>
<th>PRSTDATE</th>
<th>PRENDATE</th>
<th>MAJPROJ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type:</td>
<td>CHAR(6)</td>
<td>VARCHAR(36)</td>
<td>CHAR(3)</td>
<td>CHAR(6)</td>
<td>DEC(5,2)</td>
<td>DATE</td>
<td>DATE</td>
<td>CHAR(6)</td>
</tr>
<tr>
<td>Values:</td>
<td>AD3100</td>
<td>ADMIN SERVICES</td>
<td>D01</td>
<td>000010</td>
<td>6.50</td>
<td>01/01/1982</td>
<td>02/01/1983</td>
<td></td>
</tr>
</tbody>
</table>
The SAMPLE database

<table>
<thead>
<tr>
<th>Name:</th>
<th>PROJNO</th>
<th>PROJNAME</th>
<th>DEPTNO</th>
<th>RESPEMP</th>
<th>PRSTAFF</th>
<th>PRSDATE</th>
<th>PRSEDATE</th>
<th>MAJPROJ</th>
</tr>
</thead>
<tbody>
<tr>
<td>AD3110</td>
<td>GENERAL ADMIN SYSTEMS</td>
<td>D21</td>
<td>000070</td>
<td>6.00</td>
<td>01/01/1982</td>
<td>02/01/1983</td>
<td>AD3100</td>
<td></td>
</tr>
<tr>
<td>AD3111</td>
<td>PAYROLL PROGRAMMING</td>
<td>D21</td>
<td>000230</td>
<td>2.00</td>
<td>01/01/1982</td>
<td>02/01/1983</td>
<td>AD3100</td>
<td></td>
</tr>
<tr>
<td>AD3112</td>
<td>PERSONNEL PROGRAMMING</td>
<td>D21</td>
<td>000250</td>
<td>1.00</td>
<td>01/01/1982</td>
<td>02/01/1983</td>
<td>AD3100</td>
<td></td>
</tr>
<tr>
<td>AD3113</td>
<td>ACCOUNT PROGRAMMING</td>
<td>D21</td>
<td>000270</td>
<td>2.00</td>
<td>01/01/1982</td>
<td>02/01/1983</td>
<td>AD3100</td>
<td></td>
</tr>
<tr>
<td>IF1000</td>
<td>QUERY SERVICES</td>
<td>C01</td>
<td>000030</td>
<td>2.00</td>
<td>01/01/1982</td>
<td>02/01/1983</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IF2000</td>
<td>USER EDUCATION</td>
<td>C01</td>
<td>000030</td>
<td>1.00</td>
<td>01/01/1982</td>
<td>02/01/1983</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MA2100</td>
<td>WELD LINE AUTOMATION</td>
<td>D01</td>
<td>000010</td>
<td>12.00</td>
<td>01/01/1982</td>
<td>02/01/1983</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MA2110</td>
<td>W L PROGRAMMING</td>
<td>D11</td>
<td>000660</td>
<td>9.00</td>
<td>01/01/1982</td>
<td>02/01/1983</td>
<td>MA2100</td>
<td></td>
</tr>
<tr>
<td>MA2111</td>
<td>W L PROGRAM DESIGN</td>
<td>D11</td>
<td>00220</td>
<td>2.00</td>
<td>01/01/1982</td>
<td>12/01/1982</td>
<td>MA2100</td>
<td></td>
</tr>
<tr>
<td>MA2112</td>
<td>W L ROBOT DESIGN</td>
<td>D11</td>
<td>00150</td>
<td>3.00</td>
<td>01/01/1982</td>
<td>12/01/1982</td>
<td>MA2100</td>
<td></td>
</tr>
<tr>
<td>MA2113</td>
<td>W L PROD CONT PROGS</td>
<td>D11</td>
<td>00160</td>
<td>3.00</td>
<td>02/15/1982</td>
<td>12/01/1982</td>
<td>MA2100</td>
<td></td>
</tr>
<tr>
<td>OP1000</td>
<td>OPERATION SUPPORT</td>
<td>E01</td>
<td>00050</td>
<td>6.00</td>
<td>01/01/1982</td>
<td>02/01/1983</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OP1010</td>
<td>OPERATION</td>
<td>E11</td>
<td>00090</td>
<td>5.00</td>
<td>01/01/1982</td>
<td>02/01/1983</td>
<td>OP1000</td>
<td></td>
</tr>
<tr>
<td>OP2000</td>
<td>GEN SYSTEMS SERVICES</td>
<td>E01</td>
<td>00050</td>
<td>5.00</td>
<td>01/01/1982</td>
<td>02/01/1983</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OP2010</td>
<td>SYSTEMS SUPPORT</td>
<td>E21</td>
<td>00300</td>
<td>4.00</td>
<td>01/01/1982</td>
<td>02/01/1983</td>
<td>OP2010</td>
<td></td>
</tr>
<tr>
<td>OP2011</td>
<td>SCP SYSTEMS SUPPORT</td>
<td>E21</td>
<td>00320</td>
<td>1.00</td>
<td>01/01/1982</td>
<td>02/01/1983</td>
<td>OP2010</td>
<td></td>
</tr>
<tr>
<td>OP2012</td>
<td>APPLICATIONS SUPPORT</td>
<td>E21</td>
<td>00330</td>
<td>1.00</td>
<td>01/01/1982</td>
<td>02/01/1983</td>
<td>OP2010</td>
<td></td>
</tr>
<tr>
<td>OP2013</td>
<td>DB/DC SUPPORT</td>
<td>E21</td>
<td>00340</td>
<td>1.00</td>
<td>01/01/1982</td>
<td>02/01/1983</td>
<td>OP2010</td>
<td></td>
</tr>
<tr>
<td>PL2100</td>
<td>WELD LINE PLANNING</td>
<td>B01</td>
<td>00020</td>
<td>1.00</td>
<td>01/01/1982</td>
<td>09/15/1982</td>
<td>MA2100</td>
<td></td>
</tr>
</tbody>
</table>

PROJACT table

<table>
<thead>
<tr>
<th>Name:</th>
<th>PROJNO</th>
<th>ACTNO</th>
<th>ACSTAFF</th>
<th>ACSDATE</th>
<th>ACSENDATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type:</td>
<td>CHAR(6)</td>
<td>SMALLINT</td>
<td>DEC(5,2)</td>
<td>DATE</td>
<td>DATE</td>
</tr>
<tr>
<td>Values:</td>
<td>AD3100</td>
<td>10</td>
<td>01/01/1982</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>AD3110</td>
<td>10</td>
<td>01/01/1982</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>AD3111</td>
<td>60</td>
<td>01/01/1982</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>AD3111</td>
<td>60</td>
<td>03/15/1982</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>AD3111</td>
<td>70</td>
<td>03/15/1982</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>AD3111</td>
<td>80</td>
<td>04/15/1982</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>AD3111</td>
<td>180</td>
<td>10/15/1982</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>AD3111</td>
<td>70</td>
<td>02/15/1982</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>AD3111</td>
<td>80</td>
<td>09/15/1982</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>AD3112</td>
<td>60</td>
<td>01/01/1982</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>AD3112</td>
<td>60</td>
<td>02/01/1982</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## The SAMPLE database

<table>
<thead>
<tr>
<th>Name:</th>
<th>PROJNO</th>
<th>ACTNO</th>
<th>ACSTAFF</th>
<th>ACSDATE</th>
<th>ACEDATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>AD3112</td>
<td>60</td>
<td></td>
<td></td>
<td>01/01/1983</td>
<td></td>
</tr>
<tr>
<td>AD3112</td>
<td>70</td>
<td></td>
<td></td>
<td>02/01/1982</td>
<td></td>
</tr>
<tr>
<td>AD3112</td>
<td>70</td>
<td></td>
<td></td>
<td>03/15/1982</td>
<td></td>
</tr>
<tr>
<td>AD3112</td>
<td>70</td>
<td></td>
<td></td>
<td>08/15/1982</td>
<td></td>
</tr>
<tr>
<td>AD3112</td>
<td>80</td>
<td></td>
<td></td>
<td>08/15/1982</td>
<td></td>
</tr>
<tr>
<td>AD3112</td>
<td>80</td>
<td></td>
<td></td>
<td>10/15/1982</td>
<td></td>
</tr>
<tr>
<td>AD3112</td>
<td>180</td>
<td></td>
<td></td>
<td>08/15/1982</td>
<td></td>
</tr>
<tr>
<td>AD3113</td>
<td>70</td>
<td></td>
<td></td>
<td>06/15/1982</td>
<td></td>
</tr>
<tr>
<td>AD3113</td>
<td>70</td>
<td></td>
<td></td>
<td>07/01/1982</td>
<td></td>
</tr>
<tr>
<td>AD3113</td>
<td>80</td>
<td></td>
<td></td>
<td>01/01/1982</td>
<td></td>
</tr>
<tr>
<td>AD3113</td>
<td>80</td>
<td></td>
<td></td>
<td>03/01/1982</td>
<td></td>
</tr>
<tr>
<td>AD3113</td>
<td>180</td>
<td></td>
<td></td>
<td>03/01/1982</td>
<td></td>
</tr>
<tr>
<td>AD3113</td>
<td>180</td>
<td></td>
<td></td>
<td>04/15/1982</td>
<td></td>
</tr>
<tr>
<td>AD3113</td>
<td>180</td>
<td></td>
<td></td>
<td>06/01/1982</td>
<td></td>
</tr>
<tr>
<td>AD3113</td>
<td>60</td>
<td></td>
<td></td>
<td>03/01/1982</td>
<td></td>
</tr>
<tr>
<td>AD3113</td>
<td>60</td>
<td></td>
<td></td>
<td>04/01/1982</td>
<td></td>
</tr>
<tr>
<td>AD3113</td>
<td>60</td>
<td></td>
<td></td>
<td>09/01/1982</td>
<td></td>
</tr>
<tr>
<td>AD3113</td>
<td>70</td>
<td></td>
<td></td>
<td>09/01/1982</td>
<td></td>
</tr>
<tr>
<td>AD3113</td>
<td>70</td>
<td></td>
<td></td>
<td>10/15/1982</td>
<td></td>
</tr>
<tr>
<td>IF1000</td>
<td>10</td>
<td></td>
<td></td>
<td>06/01/1982</td>
<td></td>
</tr>
<tr>
<td>IF1000</td>
<td>90</td>
<td></td>
<td></td>
<td>10/01/1982</td>
<td></td>
</tr>
<tr>
<td>IF1000</td>
<td>100</td>
<td></td>
<td></td>
<td>10/01/1982</td>
<td></td>
</tr>
<tr>
<td>IF2000</td>
<td>10</td>
<td></td>
<td></td>
<td>01/01/1982</td>
<td></td>
</tr>
<tr>
<td>IF2000</td>
<td>100</td>
<td></td>
<td></td>
<td>01/01/1982</td>
<td></td>
</tr>
<tr>
<td>IF2000</td>
<td>100</td>
<td></td>
<td></td>
<td>03/01/1982</td>
<td></td>
</tr>
<tr>
<td>IF2000</td>
<td>110</td>
<td></td>
<td></td>
<td>03/01/1982</td>
<td></td>
</tr>
<tr>
<td>IF2000</td>
<td>110</td>
<td></td>
<td></td>
<td>10/01/1982</td>
<td></td>
</tr>
<tr>
<td>MA2100</td>
<td>10</td>
<td></td>
<td></td>
<td>01/01/1982</td>
<td></td>
</tr>
<tr>
<td>MA2100</td>
<td>20</td>
<td></td>
<td></td>
<td>01/01/1982</td>
<td></td>
</tr>
<tr>
<td>MA2110</td>
<td>10</td>
<td></td>
<td></td>
<td>01/01/1982</td>
<td></td>
</tr>
<tr>
<td>MA2111</td>
<td>50</td>
<td></td>
<td></td>
<td>01/01/1982</td>
<td></td>
</tr>
<tr>
<td>MA2111</td>
<td>60</td>
<td></td>
<td></td>
<td>06/15/1982</td>
<td></td>
</tr>
<tr>
<td>MA2111</td>
<td>40</td>
<td></td>
<td></td>
<td>01/01/1982</td>
<td></td>
</tr>
<tr>
<td>MA2112</td>
<td>60</td>
<td></td>
<td></td>
<td>01/01/1982</td>
<td></td>
</tr>
<tr>
<td>MA2112</td>
<td>180</td>
<td></td>
<td></td>
<td>07/15/1982</td>
<td></td>
</tr>
<tr>
<td>MA2112</td>
<td>70</td>
<td></td>
<td></td>
<td>06/01/1982</td>
<td></td>
</tr>
<tr>
<td>MA2112</td>
<td>70</td>
<td></td>
<td></td>
<td>01/01/1982</td>
<td></td>
</tr>
<tr>
<td>MA2112</td>
<td>80</td>
<td></td>
<td></td>
<td>10/01/1982</td>
<td></td>
</tr>
<tr>
<td>MA2113</td>
<td>60</td>
<td></td>
<td></td>
<td>07/15/1982</td>
<td></td>
</tr>
<tr>
<td>MA2113</td>
<td>80</td>
<td></td>
<td></td>
<td>01/01/1982</td>
<td></td>
</tr>
<tr>
<td>MA2113</td>
<td>70</td>
<td></td>
<td></td>
<td>04/01/1982</td>
<td></td>
</tr>
</tbody>
</table>
### The SAMPLE database

#### PROJECT table

<table>
<thead>
<tr>
<th>Name</th>
<th>PROJNO</th>
<th>ACTNO</th>
<th>ACSTAFF</th>
<th>ACSTDATE</th>
<th>ACENDATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MA2113</td>
<td>80</td>
<td>10/01/1982</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MA2113</td>
<td>180</td>
<td>10/01/1982</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OP1000</td>
<td>10</td>
<td>01/01/1982</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OP1010</td>
<td>10</td>
<td>01/01/1982</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OP1010</td>
<td>130</td>
<td>01/01/1982</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OP2010</td>
<td>10</td>
<td>01/01/1982</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OP2011</td>
<td>140</td>
<td>01/01/1982</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OP2011</td>
<td>150</td>
<td>01/01/1982</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OP2012</td>
<td>140</td>
<td>01/01/1982</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OP2012</td>
<td>160</td>
<td>01/01/1982</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OP2013</td>
<td>140</td>
<td>01/01/1982</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OP2013</td>
<td>170</td>
<td>01/01/1982</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PL2100</td>
<td>30</td>
<td>01/01/1982</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### SALES table

<table>
<thead>
<tr>
<th>Name</th>
<th>SALES_DATE</th>
<th>SALES_PERSON</th>
<th>REGION</th>
<th>SALES</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DATE</td>
<td>VARCHAR(15)</td>
<td>VARCHAR(15)</td>
<td>INTEGER</td>
</tr>
<tr>
<td>Desc</td>
<td>Date of sales</td>
<td>Employee’s last name</td>
<td>Region of sales</td>
<td>Number of sales</td>
</tr>
</tbody>
</table>
The sample database

<table>
<thead>
<tr>
<th>Name</th>
<th>SALES_DATE</th>
<th>SALES_PERSON</th>
<th>REGION</th>
<th>SALES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Values:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LUCCHESI</td>
<td>12/31/2005</td>
<td>Ontario-South</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>LUCCHESI</td>
<td>03/29/2006</td>
<td>Ontario-South</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>LUCCHESI</td>
<td>03/30/2006</td>
<td>Ontario-South</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>LUCCHESI</td>
<td>03/31/2006</td>
<td>Ontario-South</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>LUCCHESI</td>
<td>04/01/2006</td>
<td>Ontario-South</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>LUCCHESI</td>
<td>04/01/2006</td>
<td>Ontario-South</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>LUCCHESI</td>
<td>04/01/2006</td>
<td>Ontario-North</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>LUCCHESI</td>
<td>04/01/2006</td>
<td>Ontario-North</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>GOUNOT</td>
<td>03/29/2006</td>
<td>Quebec</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>GOUNOT</td>
<td>03/30/2006</td>
<td>Quebec</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>GOUNOT</td>
<td>04/01/2006</td>
<td>Ontario-South</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>GOUNOT</td>
<td>04/01/2006</td>
<td>Ontario-South</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>GOUNOT</td>
<td>04/01/2006</td>
<td>Ontario-North</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>GOUNOT</td>
<td>04/01/2006</td>
<td>Quebec</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>GOUNOT</td>
<td>04/01/2006</td>
<td>Manitoba</td>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>

Appendix F. The sample database 1015
The SAMPLE database

**STAFF table**

<table>
<thead>
<tr>
<th>ID</th>
<th>NAME</th>
<th>DEPT</th>
<th>JOB</th>
<th>YEARS</th>
<th>SALARY</th>
<th>COMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Sanders</td>
<td>20</td>
<td>Mgr</td>
<td>7</td>
<td>16352.50</td>
<td>-</td>
</tr>
<tr>
<td>20</td>
<td>Pernal</td>
<td>20</td>
<td>Sales</td>
<td>8</td>
<td>18171.25</td>
<td>612.45</td>
</tr>
<tr>
<td>30</td>
<td>Marenghi</td>
<td>38</td>
<td>Mgr</td>
<td>5</td>
<td>17506.75</td>
<td>-</td>
</tr>
<tr>
<td>40</td>
<td>O'Brien</td>
<td>38</td>
<td>Sales</td>
<td>6</td>
<td>18006.00</td>
<td>846.55</td>
</tr>
<tr>
<td>50</td>
<td>Quigley</td>
<td>38</td>
<td>Sales</td>
<td>7</td>
<td>16808.30</td>
<td>650.25</td>
</tr>
<tr>
<td>60</td>
<td>Rothman</td>
<td>15</td>
<td>Clerk</td>
<td>5</td>
<td>12954.75</td>
<td>180.00</td>
</tr>
<tr>
<td>70</td>
<td>Yarnaguchi</td>
<td>42</td>
<td>Clerk</td>
<td>6</td>
<td>10505.90</td>
<td>75.60</td>
</tr>
<tr>
<td>80</td>
<td>Frays</td>
<td>51</td>
<td>Mgr</td>
<td>6</td>
<td>21350.00</td>
<td>-</td>
</tr>
<tr>
<td>90</td>
<td>Williams</td>
<td>51</td>
<td>Sales</td>
<td>5</td>
<td>19490.50</td>
<td>637.00</td>
</tr>
<tr>
<td>100</td>
<td>Molina</td>
<td>10</td>
<td>Mgr</td>
<td>7</td>
<td>22099.20</td>
<td>-</td>
</tr>
<tr>
<td>110</td>
<td>Kruske</td>
<td>15</td>
<td>Clerk</td>
<td>4</td>
<td>12258.50</td>
<td>110.50</td>
</tr>
<tr>
<td>120</td>
<td>Abrahams</td>
<td>36</td>
<td>Clerk</td>
<td>3</td>
<td>12309.75</td>
<td>236.50</td>
</tr>
<tr>
<td>130</td>
<td>Sneedor</td>
<td>20</td>
<td>Clerk</td>
<td>3</td>
<td>14252.75</td>
<td>126.50</td>
</tr>
<tr>
<td>140</td>
<td>Schollen</td>
<td>42</td>
<td>Clerk</td>
<td>4</td>
<td>11508.60</td>
<td>84.25</td>
</tr>
<tr>
<td>150</td>
<td>Lee</td>
<td>10</td>
<td>Mgr</td>
<td>10</td>
<td>20521.00</td>
<td>-</td>
</tr>
<tr>
<td>160</td>
<td>Moira</td>
<td>51</td>
<td>Sales</td>
<td>7</td>
<td>17054.50</td>
<td>992.80</td>
</tr>
<tr>
<td>170</td>
<td>Ludgquist</td>
<td>51</td>
<td>Clerk</td>
<td>2</td>
<td>13509.80</td>
<td>589.05</td>
</tr>
<tr>
<td>180</td>
<td>Daniels</td>
<td>10</td>
<td>Mgr</td>
<td>5</td>
<td>19204.25</td>
<td>-</td>
</tr>
<tr>
<td>190</td>
<td>Wonder</td>
<td>51</td>
<td>Clerk</td>
<td>6</td>
<td>14461.00</td>
<td>513.50</td>
</tr>
<tr>
<td>200</td>
<td>Jones</td>
<td>10</td>
<td>Mgr</td>
<td>12</td>
<td>21234.00</td>
<td>-</td>
</tr>
<tr>
<td>210</td>
<td>Lee</td>
<td>66</td>
<td>Mgr</td>
<td>9</td>
<td>10505.50</td>
<td>-</td>
</tr>
<tr>
<td>220</td>
<td>Wilson</td>
<td>66</td>
<td>Sales</td>
<td>9</td>
<td>16074.75</td>
<td>831.75</td>
</tr>
<tr>
<td>230</td>
<td>Quill</td>
<td>84</td>
<td>Mgr</td>
<td>10</td>
<td>19188.00</td>
<td>-</td>
</tr>
<tr>
<td>240</td>
<td>Davis</td>
<td>84</td>
<td>Sales</td>
<td>5</td>
<td>15494.50</td>
<td>806.10</td>
</tr>
<tr>
<td>250</td>
<td>Graham</td>
<td>66</td>
<td>Sales</td>
<td>13</td>
<td>21088.00</td>
<td>202.70</td>
</tr>
<tr>
<td>260</td>
<td>Gourley</td>
<td>66</td>
<td>Sales</td>
<td>4</td>
<td>16955.20</td>
<td>544.00</td>
</tr>
<tr>
<td>270</td>
<td>Burke</td>
<td>66</td>
<td>Clerk</td>
<td>1</td>
<td>10868.00</td>
<td>53.50</td>
</tr>
<tr>
<td>280</td>
<td>Edwards</td>
<td>84</td>
<td>Sales</td>
<td>7</td>
<td>17084.00</td>
<td>2683.00</td>
</tr>
<tr>
<td>290</td>
<td>Cufney</td>
<td>84</td>
<td>Clerk</td>
<td>5</td>
<td>13351.50</td>
<td>188.00</td>
</tr>
</tbody>
</table>

**PRODUCT table**

<table>
<thead>
<tr>
<th>PID</th>
<th>NAME</th>
<th>PRICE</th>
<th>PROMOPRICE</th>
<th>PROMOSTART</th>
<th>PROMOEND</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>Snow Shovel, Basic 22 inch</td>
<td>9.99</td>
<td>7.25</td>
<td>11/19/2004</td>
<td>12/19/2004</td>
<td>p1.xml</td>
</tr>
<tr>
<td>100</td>
<td>Snow Shovel, Deluxe 24 inch</td>
<td>19.99</td>
<td>15.99</td>
<td>12/18/2005</td>
<td>02/28/2006</td>
<td>p2.xml</td>
</tr>
<tr>
<td>100</td>
<td>Ice Scraper, Windshield 4 inch</td>
<td>3.99</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>p4.xml</td>
</tr>
</tbody>
</table>

Here is the XML schema definition file for the XML column in the above table: product.xsd

**PURCHASEORDER table**

<table>
<thead>
<tr>
<th>POID</th>
<th>STATUS</th>
<th>CUSTID</th>
<th>ORDERDATE</th>
<th>ORDERER</th>
<th>COMMENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>5000</td>
<td>Unshipped</td>
<td>1002</td>
<td>02/18/2006</td>
<td>po1.xml</td>
<td>THIS IS A NEW PURCHASE ORDER</td>
</tr>
<tr>
<td>5001</td>
<td>Shipped</td>
<td>1003</td>
<td>02/03/2005</td>
<td>po2.xml</td>
<td>THIS IS A NEW PURCHASE ORDER</td>
</tr>
<tr>
<td>5002</td>
<td>Shipped</td>
<td>1001</td>
<td>02/28/2004</td>
<td>po3.xml</td>
<td>THIS IS A NEW PURCHASE ORDER</td>
</tr>
</tbody>
</table>
Here is the XML schema definition file for the XML column in the above table: `porder.xsd`

**CUSTOMER table**

<table>
<thead>
<tr>
<th>Name: CID</th>
<th>INFO</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type: BIGINT Not Null</td>
<td>XML</td>
</tr>
<tr>
<td>Values: 1000</td>
<td>c1.xml</td>
</tr>
<tr>
<td>1001</td>
<td>c2.xml</td>
</tr>
<tr>
<td>1002</td>
<td>c3.xml</td>
</tr>
<tr>
<td>1003</td>
<td>c4.xml</td>
</tr>
<tr>
<td>1004</td>
<td>c5.xml</td>
</tr>
<tr>
<td>1005</td>
<td>c6.xml</td>
</tr>
</tbody>
</table>

Here is the XML schema definition file for the XML column in the above table: `customer.xsd`

**CATALOG table**

<table>
<thead>
<tr>
<th>Name: NAME</th>
<th>CATALOG</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type: VARCHAR(128) Not Null</td>
<td>XML</td>
</tr>
<tr>
<td>Values: Spring catalog</td>
<td>cat1.xml</td>
</tr>
<tr>
<td>Summer catalog</td>
<td>cat2.xml</td>
</tr>
<tr>
<td>Fall catalog</td>
<td>cat3.xml</td>
</tr>
<tr>
<td>Winter catalog</td>
<td>cat4.xml</td>
</tr>
</tbody>
</table>

Here is the XML schema definition file for the XML column in the above table: `catalog.xsd`

**INVENTORY table**

<table>
<thead>
<tr>
<th>Name: PID</th>
<th>QUANTITY</th>
<th>LOCATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type: VARCHAR(10) Not Null</td>
<td>INTEGER</td>
<td>VARCHAR(128)</td>
</tr>
<tr>
<td>Values: 100-100-01</td>
<td>5</td>
<td>--</td>
</tr>
<tr>
<td>100-101-01</td>
<td>25</td>
<td>Store</td>
</tr>
<tr>
<td>100-103-01</td>
<td>55</td>
<td>Store</td>
</tr>
<tr>
<td>100-201-01</td>
<td>99</td>
<td>Warehouse</td>
</tr>
</tbody>
</table>
The SAMPLE database

### PRODUCTSUPPLIER table

<table>
<thead>
<tr>
<th>Name</th>
<th>PID</th>
<th>SID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>VARCHAR(10) Not Null</td>
<td>VARCHAR(10) Not Null</td>
</tr>
<tr>
<td>Values</td>
<td>100-100-01</td>
<td>123-456-78</td>
</tr>
<tr>
<td></td>
<td>100-101-01</td>
<td>123-456-78</td>
</tr>
<tr>
<td></td>
<td>100-103-01</td>
<td>555-789-00</td>
</tr>
<tr>
<td></td>
<td>100-201-01</td>
<td>989-897-23</td>
</tr>
</tbody>
</table>

### SUPPLIERS table

<table>
<thead>
<tr>
<th>Name</th>
<th>SID</th>
<th>ADDR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>VARCHAR(10) Not Null</td>
<td>XML</td>
</tr>
<tr>
<td>Values</td>
<td>123-456-78</td>
<td>s1.xml</td>
</tr>
<tr>
<td></td>
<td>555-789-00</td>
<td>s2.xml</td>
</tr>
<tr>
<td></td>
<td>989-897-23</td>
<td>s3.xml</td>
</tr>
<tr>
<td></td>
<td>111-898-45</td>
<td>s4.xml</td>
</tr>
</tbody>
</table>

Here is the XML schema definition file for the XML column in the above table:
supplier.xsd

**Sample files with BLOB and CLOB data type**

This section shows the data found in the EMP_PHOTO files (pictures of employees) and EMP_RESUME files (resumes of employees).

**Quintana photo**

![Quintana photo](Figure 14. Dolores M. Quintana)

**Quintana resume**

The following text is found in the file db200130.asc.

Resume: Dolores M. Quintana

Personal Information

Address:
1150 Eglinton Ave Mellonville, Idaho 83725

Phone:
(208) 555-9933
Birthdate: September 15, 1925
Sex: Female
Marital Status: Married
Height: 5'2"
Weight: 120 lbs.

Department Information
Employee Number: 000130
Dept Number: C01
Manager: Sally Kwan
Position: Analyst
Phone: (208) 555-4578
Hire Date: 1971-07-28

Education
1965 Math and English, B.A. Adelphi University
1960 Dental Technician Florida Institute of Technology

Work History
10/91 - present
Advisory Systems Analyst Producing documentation tools for engineering department.

12/85 - 9/91
Technical Writer, Writer, text programmer, and planner.

1/79 - 11/85
COBOL Payroll Programmer Writing payroll programs for a diesel fuel company.

Interests
• Cooking
• Reading
• Sewing
• Remodeling

Following is the contents of the file db200130.htm.

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 3//EN">
<html>
<head>
<title>Resume: Delores M. Quintana</title>
</head>
<body>

The SAMPLE database

Birthdate: September 15, 1925
Sex: Female
Marital Status: Married
Height: 5'2"
Weight: 120 lbs.

Department Information
Employee Number: 000130
Dept Number: C01
Manager: Sally Kwan
Position: Analyst
Phone: (208) 555-4578
Hire Date: 1971-07-28

Education
1965 Math and English, B.A. Adelphi University
1960 Dental Technician Florida Institute of Technology

Work History
10/91 - present
Advisory Systems Analyst Producing documentation tools for engineering department.

12/85 - 9/91
Technical Writer, Writer, text programmer, and planner.

1/79 - 11/85
COBOL Payroll Programmer Writing payroll programs for a diesel fuel company.

Interests
• Cooking
• Reading
• Sewing
• Remodeling

Following is the contents of the file db200130.htm.
Resume: Delores M. Quintana

Personal Information

Address: 1150 Eglinton Ave
Mellonville, Idaho 83725
Phone: (208) 875-9933
Birthdate: September 15, 1925
Sex: Female
Marital Status: Married
Height: 5'2"
Weight: 120 lbs.

Department Information

Employee Number: 000130
Dept Number: C01
Manager: Sally Kwan
Position: Analyst
Phone: (208) 385-4578
Hire Date: 1971-07-28

Education

1965
Math and English, B.A.
Adelphi University
1960
Dental Technician
Florida Institute of Technology

Work History

10/91 - present
Advisory Systems Analyst
Producing documentation tools for engineering department.
12/85 - 9/91
Technical Writer
1/79 - 11/85
COBOL Payroll Programmer

Writing payroll programs for a diesel fuel company.

Interests
- Cooking
- Reading
- Sewing
- Remodeling

Nicholls photo

Figure 15. Heather A. Nicholls

Nicholls resume

The following text is found in the file db200140.asc.

Resume: Heather A. Nicholls

Personal Information

Address:
844 Don Mills Ave Mellonville, Idaho 83734

Phone:
(208) 555-2310

Birthdate:
January 19, 1946

Sex: Female

Marital Status:
Single

Height:
5'8"

Weight:
130 lbs.

Department Information

Employee Number:
000140

Dept Number:
C01

Manager:
Sally Kwan
Position: Analyst

Phone: (208) 555-1793

Hire Date: 1976-12-15

Education

1972 Computer Engineering, Ph.D. University of Washington
1969 Music and Physics, M.A. Vassar College

Work History

2/83 - present
Architect, OCR Development Designing the architecture of OCR products.

12/76 - 1/83
Text Programmer Optical CHARacter recognition (OCR) programming in PL/I.

9/72 - 11/76
Punch Card Quality Analyst Checking punch cards met quality specifications.

Interests

• Model railroading
• Interior decorating
• Embroidery
• Knitting

Following is the content of the file db200140.htm.
<DT>Birthdate: <DD>January 19, 1946
<DT>Sex: <DD>Female
<DT>Marital Status: <DD>Single
<DT>Height: <DD>5'8"
<DT>Weight: <DD>130 lbs.<P>

<H3><A NAME="Header_3">Department Information</A></H3>
<DL COMPACT>
<DT>Employee Number: <DD>000140
<DT>Dept Number: <DD>C01
<DT>Manager: <DD>Sally Kwan
<DT>Position: <DD>Analyst
<DT>Phone: <DD>(208) 385-1793
<DT>Hire Date: <DD>1976-12-15<P>

<H3><A NAME="Header_4">Education</A></H3>
<DL>
<DT>1972 <DD>Computer Engineering, Ph.D.
University of Washington
<DT>1969 <DD>Music and Physics, B.A.
Vassar College
</DL>

<H3><A NAME="Header_5">Work History</A></H3>
<DL>
<DT>2/83 - present <DD>Architect, OCR Development
Designing the architecture of OCR products.
<DT>12/76 - 1/83 <DD>Text Programmer
Optical CHARacter recognition (OCR) programming in PL/I.
<DT>9/72 - 11/76 <DD>Punch Card Quality Analyst
Checking punch cards met quality specifications.<P>

<H3><A NAME="Header_6">Interests</A></H3>
<UL COMPACT>
<LI>Model railroading
<LI>Interior decorating
<LI>Embroidery
<LI>Knitting
</UL>
The SAMPLE database

Adamson photo

Figure 16. Bruce Adamson

Adamson resume

The following text is found in the file db200150.asc.

Resume: Bruce Adamson

Personal Information

Address:
3600 Steeles Ave Mellonville, Idaho 83757

Phone:
(208) 555-4489

Birthdate:
May 17, 1947

Sex:
Male

Marital Status:
Married

Height:
6’0”

Weight:
175 lbs.

Department Information

Employee Number:
000150

Dept Number:
D11

Manager:
Irving Stern

Position:
Designer

Phone:
(208) 555-4510

Hire Date:
1972-02-12
Education
1968  American History, B.A. Northwestern University

Work History
8/79 - present
   Neural Network Design Developing neural networks for machine intelligence products.

2/72 - 7/79
   Robot Vision Development Developing rule-based systems to emulate sight.

9/71 - 1/72
   Numerical Integration Specialist Helping bank systems communicate with each other.

Interests
- Racing motorcycles
- Building loudspeakers
- Assembling personal computers
- Sketching

Following is the content of the file db200150.htm.
The SAMPLE database

Weight: 175 lbs.

Department Information

Employee Number: 000150
Dept Number: D11
Manager: Irving Stern
Position: Designer
Phone: (208) 385-4510
Hire Date: 1972-02-12

Education

1971
Environmental Engineering, M.Sc.
Johns Hopkins University
1968
American History, B.A.
Northwestern University

Work History

8/79 - present
Developing neural networks for machine intelligence products.
Neural Network Design

2/72 - 7/79
Developing rule-based systems to emulate sight.
Robot Vision Development

9/71 - 1/72
Helping bank systems communicate with each other.
Numerical Integration Specialist

Interests

Racing motorcycles
Building loudspeakers
Assembling personal computers
Sketching

Walker photo

Figure 17. James H. Walker

Walker resume

The following text is found in the file db200190.asc.

Resume: James H. Walker
The SAMPLE database

Personal Information
Address:
3500 Steeles Ave Mellonville, Idaho 83757
Phone:
(208) 555-7325
Birthdate:
June 25, 1952
Sex:
Male
Marital Status:
Single
Height:
5’11”
Weight:
166 lbs.

Department Information
Employee Number:
000190
Dept Number:
D11
Manager:
Irving Stern
Position:
Designer
Phone:
(208) 555-2986
Hire Date:
1974-07-26

Education
1974  Computer Studies, B.Sc. University of Massachusetts
1972  Linguistic Anthropology, B.A. University of Toronto

Work History
6/87 - present
Microcode Design Optimizing algorithms for mathematical functions.
4/77 - 5/87
Printer Technical Support Installing and supporting laser printers.
9/74 - 3/77
Maintenance Programming Patching assembly language compiler for mainframes.

Interests
• Wine tasting
• Skiing
• Swimming
The SAMPLE database

- Dancing

Following is the content of the file db200190.htm.
Optimizing algorithms for mathematical functions.

4/77 - 5/87
Printer Technical Support
Installing and supporting laser printers.

9/74 - 3/77
Maintenance Programming
Patching assembly language compiler for mainframes.

Interests
- Wine tasting
- Skiing
- Swimming
- Dancing
Appendix G. Reserved schema names and reserved words

There are restrictions on the use of certain names that are required by the database manager. In some cases, names are reserved, and cannot be used by application programs. In other cases, certain names are not recommended for use by application programs, although their use is not prevented by the database manager.

The reserved schema names are:
- SYSCAT
- SYSFUN
- SYSIBM
- SYSIBMADM
- SYSPROC
- SYSPUBLIC
- SYSSTAT

It is strongly recommended that schema names never begin with the ‘SYS’ prefix, because ‘SYS’, by convention, is used to indicate an area that is reserved by the system. No aliases, global variables, triggers, user-defined functions, or user-defined types can be placed into a schema whose name starts with ‘SYS’ (SQLSTATE 42939).

The DB2QP schema and the SYSTOOLS schema are set aside for use by DB2 tools. It is recommended that users not explicitly define objects in these schemas, although their use is not prevented by the database manager.

It is also recommended that SESSION not be used as a schema name. Because declared temporary tables must be qualified by SESSION, it is possible to have an application declare a temporary table with a name that is identical to that of a persistent table, complicating the application logic. To avoid this possibility, do not use the schema SESSION except when dealing with declared temporary tables.

There are no specifically reserved words in DB2 Version 9. Keywords can be used as ordinary identifiers, except in a context where they could also be interpreted as SQL keywords. In such cases, the word must be specified as a delimited identifier. For example, COUNT cannot be used as a column name in a SELECT statement, unless it is delimited.

ISO/ANSI SQL2003 and other DB2 database products include reserved words that are not enforced by DB2 Database for Linux, UNIX, and Windows; however, it is recommended that these words not be used as ordinary identifiers, because it reduces portability.

For portability across the DB2 database products, the following should be considered reserved words:

- ACTIVATE
- ADD
- AFTER
- ALIAS
- ALL
- ALLOCATE
- DOCUMENT
- DOUBLE
- DROP
- DSSIZE
- DYNAMIC
- EACH
- LOCK
- LOCKMAX
- LOCKSIZE
- LONG
- LOOP
- MAINTAINED
- ROUND_CEILING
- ROUND_DOWN
- ROUND_FLOOR
- ROUND_HALF_DOWN
- ROUND_HALF_EVEN
- ROUND_HALF_UP
Reserved schema names and reserved words

<table>
<thead>
<tr>
<th>Allow</th>
<th>Editproc</th>
<th>Materialized</th>
<th>Round_up</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alter</td>
<td>Else</td>
<td>Maxvalue</td>
<td>Routine</td>
</tr>
<tr>
<td>And</td>
<td>Elseif</td>
<td>Microsecond</td>
<td>Row</td>
</tr>
<tr>
<td>Any</td>
<td>Enable</td>
<td>Microseconds</td>
<td>Row_number</td>
</tr>
<tr>
<td>As</td>
<td>Encoding</td>
<td>Minutes</td>
<td>Rownumber</td>
</tr>
<tr>
<td>ASensitive</td>
<td>Encryption</td>
<td>Minutes</td>
<td>Rows</td>
</tr>
<tr>
<td>Associate</td>
<td>End</td>
<td>Minvalue</td>
<td>Rowset</td>
</tr>
<tr>
<td>ASubmit</td>
<td>End-Exec</td>
<td>Mode</td>
<td>Run</td>
</tr>
<tr>
<td>At</td>
<td>Ending</td>
<td>Modifies</td>
<td>Savepoint</td>
</tr>
<tr>
<td>Attributes</td>
<td>Erase</td>
<td>Month</td>
<td>Schema</td>
</tr>
<tr>
<td>Audit</td>
<td>Escape</td>
<td>Monthss</td>
<td>Scratchpad</td>
</tr>
<tr>
<td>Authorization</td>
<td>Every</td>
<td>Nan</td>
<td>Scroll</td>
</tr>
<tr>
<td>Aux</td>
<td>Except</td>
<td>New</td>
<td>Search</td>
</tr>
<tr>
<td>Auxiliary</td>
<td>Exception</td>
<td>New_table</td>
<td>Session</td>
</tr>
<tr>
<td>Before</td>
<td>Excluding</td>
<td>Nextval</td>
<td>Session_user</td>
</tr>
<tr>
<td>Begin</td>
<td>Exclusive</td>
<td>No</td>
<td>Set</td>
</tr>
<tr>
<td>Between</td>
<td>Execute</td>
<td>Nocache</td>
<td>Seqcty</td>
</tr>
<tr>
<td>Binary</td>
<td>Exists</td>
<td>Nocycle</td>
<td>Security</td>
</tr>
<tr>
<td>Bufferpool</td>
<td>Exit</td>
<td>Nodename</td>
<td>Select</td>
</tr>
<tr>
<td>By</td>
<td>Explain</td>
<td>Nodename</td>
<td>Sequence</td>
</tr>
<tr>
<td>Cache</td>
<td>External</td>
<td>Nomaxvalue</td>
<td>Sequence</td>
</tr>
<tr>
<td>Call</td>
<td>Extract</td>
<td>Nominvalue</td>
<td>Sequence</td>
</tr>
<tr>
<td>Called</td>
<td>Fenced</td>
<td>None</td>
<td>Session_user</td>
</tr>
<tr>
<td>Capture</td>
<td>Fetch</td>
<td>Noorder</td>
<td>Set</td>
</tr>
<tr>
<td>Cardinality</td>
<td>Fieldproc</td>
<td>Normalized</td>
<td>Signal</td>
</tr>
<tr>
<td>Cased</td>
<td>File</td>
<td>Not</td>
<td>Simple</td>
</tr>
<tr>
<td>Case</td>
<td>Final</td>
<td>Null</td>
<td>Snan</td>
</tr>
<tr>
<td>Cast</td>
<td>For</td>
<td>Nulls</td>
<td>Some</td>
</tr>
<tr>
<td>Ccsid</td>
<td>Foreign</td>
<td>Numparts</td>
<td>Source</td>
</tr>
<tr>
<td>Char</td>
<td>Free</td>
<td>Obid</td>
<td>Specific</td>
</tr>
<tr>
<td>Character</td>
<td>From</td>
<td>Of</td>
<td>Sql</td>
</tr>
<tr>
<td>Check</td>
<td>Full</td>
<td>Old</td>
<td>Sqli</td>
</tr>
<tr>
<td>Clone</td>
<td>Function</td>
<td>Old_table</td>
<td>Stacked</td>
</tr>
<tr>
<td>Close</td>
<td>General</td>
<td>On</td>
<td>Standard</td>
</tr>
<tr>
<td>Cluster</td>
<td>Generated</td>
<td>Open</td>
<td>Start</td>
</tr>
<tr>
<td>Collection</td>
<td>Get</td>
<td>Optimization</td>
<td>Starting</td>
</tr>
<tr>
<td>Collid</td>
<td>Global</td>
<td>Optimize</td>
<td>Statement</td>
</tr>
<tr>
<td>Column</td>
<td>Go</td>
<td>Option</td>
<td>Static</td>
</tr>
<tr>
<td>Comment</td>
<td>Goto</td>
<td>Or</td>
<td>Statment</td>
</tr>
<tr>
<td>Commit</td>
<td>Grant</td>
<td>Order</td>
<td>Stay</td>
</tr>
<tr>
<td>Concat</td>
<td>Graphic</td>
<td>Out</td>
<td>Stogroup</td>
</tr>
<tr>
<td>Condition</td>
<td>Group</td>
<td>Outer</td>
<td>Stores</td>
</tr>
<tr>
<td>Connect</td>
<td>Handler</td>
<td>Over</td>
<td>Style</td>
</tr>
<tr>
<td>Connection</td>
<td>Hash</td>
<td>Overriding</td>
<td>Substring</td>
</tr>
<tr>
<td>Constraint</td>
<td>Hashtag_value</td>
<td>Package</td>
<td>Summary</td>
</tr>
<tr>
<td>Contains</td>
<td>Having</td>
<td>Padded</td>
<td>Synonym</td>
</tr>
<tr>
<td>Continue</td>
<td>Hint</td>
<td>Pagesize</td>
<td>Synfun</td>
</tr>
<tr>
<td>Count</td>
<td>Hold</td>
<td>Parameter</td>
<td>Synbm</td>
</tr>
<tr>
<td>Count_big</td>
<td>Hour</td>
<td>Part</td>
<td>Sysproc</td>
</tr>
<tr>
<td>Create</td>
<td>Hours</td>
<td>Partition</td>
<td>System</td>
</tr>
<tr>
<td>Cross</td>
<td>Identity</td>
<td>Partitioned</td>
<td>System_user</td>
</tr>
<tr>
<td>Current</td>
<td>If</td>
<td>Partitioning</td>
<td>Table</td>
</tr>
<tr>
<td>Current_date</td>
<td>Immediate</td>
<td>Partitions</td>
<td>Tablespace</td>
</tr>
<tr>
<td>Current_lcctype</td>
<td>Including</td>
<td>Path</td>
<td>Then</td>
</tr>
<tr>
<td>Current_path</td>
<td>Inclusive</td>
<td>Piecesize</td>
<td>Time</td>
</tr>
<tr>
<td>Current_schema</td>
<td>Increment</td>
<td>Plan</td>
<td>To</td>
</tr>
<tr>
<td>Current_server</td>
<td>Index</td>
<td>Position</td>
<td>Transaction</td>
</tr>
<tr>
<td>Current_timestamp</td>
<td>Indicator</td>
<td>Precision</td>
<td>Trigger</td>
</tr>
<tr>
<td>Current_timezone</td>
<td>Inf</td>
<td>Prepare</td>
<td>Trim</td>
</tr>
<tr>
<td>Current_user</td>
<td>Infinity</td>
<td>Prevval</td>
<td>Truncate</td>
</tr>
<tr>
<td>Cursor</td>
<td>Inherit</td>
<td>Primary</td>
<td>Type</td>
</tr>
<tr>
<td>Cycle</td>
<td>Inner</td>
<td>Priqty</td>
<td>Undo</td>
</tr>
<tr>
<td>Data</td>
<td>Inout</td>
<td>Privileges</td>
<td>Unique</td>
</tr>
<tr>
<td>Database</td>
<td>InSENSITIVE</td>
<td>Procedure</td>
<td>Until</td>
</tr>
<tr>
<td>Datapartitionname</td>
<td>Insert</td>
<td>Program</td>
<td>Update</td>
</tr>
<tr>
<td>Datapartitionnum</td>
<td>Integrity</td>
<td>Psid</td>
<td></td>
</tr>
</tbody>
</table>
Reserved schema names and reserved words

<table>
<thead>
<tr>
<th>DATE</th>
<th>INTERSECT</th>
<th>PUBLIC</th>
<th>USAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>DAY</td>
<td>INTO</td>
<td>QUERY</td>
<td>USER</td>
</tr>
<tr>
<td>DAYS</td>
<td>IS</td>
<td>QUERYNO</td>
<td>USING</td>
</tr>
<tr>
<td>DB2GENERAL</td>
<td>ISOBID</td>
<td>RANGE</td>
<td>VALIDPROC</td>
</tr>
<tr>
<td>DB2GENRL</td>
<td>ISOLATION</td>
<td>RANK</td>
<td>VALUE</td>
</tr>
<tr>
<td>DB2SQL</td>
<td>ITERATE</td>
<td>READ</td>
<td>VALUES</td>
</tr>
<tr>
<td>DBINFO</td>
<td>JAR</td>
<td>READS</td>
<td>VARIABLE</td>
</tr>
<tr>
<td>DBPARTITIONNAME</td>
<td>JAVA</td>
<td>RECOVERY</td>
<td>VARIANT</td>
</tr>
<tr>
<td>DBPARTITIONNUM</td>
<td>JOIN</td>
<td>REFERENCES</td>
<td>VCAT</td>
</tr>
<tr>
<td>DEALLOCATE</td>
<td>KEEP</td>
<td>REFERENCING</td>
<td>VERSION</td>
</tr>
<tr>
<td>DECLARE</td>
<td>KEY</td>
<td>REFRESH</td>
<td>VIEW</td>
</tr>
<tr>
<td>DEFAULT</td>
<td>LABEL</td>
<td>RELEASE</td>
<td>VOLATILE</td>
</tr>
<tr>
<td>DEFAULTS</td>
<td>LANGUAGE</td>
<td>RENAME</td>
<td>VOLUMES</td>
</tr>
<tr>
<td>DEFINITION</td>
<td>LATERAL</td>
<td>REPEAT</td>
<td>WHEN</td>
</tr>
<tr>
<td>DELETE</td>
<td>LC_TYPE</td>
<td>RESET</td>
<td>WHENEVER</td>
</tr>
<tr>
<td>DENSE_RANK</td>
<td>LEFT</td>
<td>RESIGNAL</td>
<td>WHERE</td>
</tr>
<tr>
<td>DENSERANK</td>
<td>LIKE</td>
<td>RESTART</td>
<td>WHILE</td>
</tr>
<tr>
<td>DESCRIPTOR</td>
<td>LINKTYPE</td>
<td>RESULT</td>
<td>WITHOUT</td>
</tr>
<tr>
<td>DETERMINISTIC</td>
<td>LOCAL</td>
<td>RESULT_SET_LOCATOR</td>
<td>WLM</td>
</tr>
<tr>
<td>DIAGNOSTICS</td>
<td>LOCALDATE</td>
<td>RETURN</td>
<td>WRITE</td>
</tr>
<tr>
<td>DISABLE</td>
<td>LOCAL</td>
<td>RETURNS</td>
<td>XML_ELEMENT</td>
</tr>
<tr>
<td>DISALLOW</td>
<td>LOCALTIME</td>
<td>REVOKE</td>
<td>XML exists</td>
</tr>
<tr>
<td>DISCONNECT</td>
<td>LOCALTIMESTAMP</td>
<td>RIGHT</td>
<td>XML NAMESPACEs</td>
</tr>
<tr>
<td>DISTINCT</td>
<td>LOCATOR</td>
<td>ROLE</td>
<td>YEAR</td>
</tr>
<tr>
<td>DO</td>
<td>LOCATORS</td>
<td>ROLLBACK</td>
<td>YEARS</td>
</tr>
</tbody>
</table>

The following list contains the ISO/ANSI SQL2003 reserved words that are not in the previous list:

<table>
<thead>
<tr>
<th>ABS</th>
<th>GROUPING</th>
<th>REGR_INTERCEPT</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARE</td>
<td>INT</td>
<td>REGR_R2</td>
</tr>
<tr>
<td>ARRAY</td>
<td>INTEGER</td>
<td>REGR_SLOPE</td>
</tr>
<tr>
<td>ASYMMETRIC</td>
<td>INTERSECTION</td>
<td>REGR_SXX</td>
</tr>
<tr>
<td>ATOMIC</td>
<td>INTERVAL</td>
<td>REGR_SXY</td>
</tr>
<tr>
<td>AVG</td>
<td>LARGE</td>
<td>REGR_SYY</td>
</tr>
<tr>
<td>BIGINT</td>
<td>LEADING</td>
<td>ROLLUP</td>
</tr>
<tr>
<td>BLOB</td>
<td>LN</td>
<td>SCOPE</td>
</tr>
<tr>
<td>BOOLEAN</td>
<td>LOWER</td>
<td>SIMILAR</td>
</tr>
<tr>
<td>BOTH</td>
<td>MATCH</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>CEIL</td>
<td>MAX</td>
<td>SPECIFICITY</td>
</tr>
<tr>
<td>CEILING</td>
<td>MEMBER</td>
<td>SQLEXCEPTION</td>
</tr>
<tr>
<td>CHAR_LENGTH</td>
<td>MERGE</td>
<td>SQLSTATE</td>
</tr>
<tr>
<td>CHARACTER_LENGTH</td>
<td>METHOD</td>
<td>SQLWARNING</td>
</tr>
<tr>
<td>CLOB</td>
<td>MIN</td>
<td>SORT</td>
</tr>
<tr>
<td>COALESCE</td>
<td>MOD</td>
<td>STDDEV_POP</td>
</tr>
<tr>
<td>COLLATE</td>
<td>MODULE</td>
<td>STDDEV_SAMP</td>
</tr>
<tr>
<td>COLLECT</td>
<td>MULTISET</td>
<td>SUBMULTISET</td>
</tr>
<tr>
<td>CONVERT</td>
<td>NATIONAL</td>
<td>SUM</td>
</tr>
<tr>
<td>CORR</td>
<td>NATURAL</td>
<td>SYMMETRIC</td>
</tr>
<tr>
<td>CORRESPONDING</td>
<td>NCHAR</td>
<td>TABLESAMPLE</td>
</tr>
<tr>
<td>COVAR_POP</td>
<td>NCHAR</td>
<td>TIMEZONE_HOUR</td>
</tr>
<tr>
<td>COVAR_SAMP</td>
<td>NORMALIZE</td>
<td>TIMEZONE_MINUTE</td>
</tr>
<tr>
<td>CUBE</td>
<td>NULLIF</td>
<td>TRAILING</td>
</tr>
<tr>
<td>CUME_DIST</td>
<td>NUMERIC</td>
<td>TRANSLATE</td>
</tr>
<tr>
<td>CURRENT_DEFAULT_TRANSFORM_GROUP</td>
<td>OCTET_LENGTH</td>
<td>TRANSLATION</td>
</tr>
<tr>
<td>CURRENT_ROLE</td>
<td>ONLY</td>
<td>TREAT</td>
</tr>
<tr>
<td>CURRENT_TRANSFORM_GROUP_FOR_TYPE</td>
<td>OVERLAPS</td>
<td>TRUE</td>
</tr>
<tr>
<td>DEC</td>
<td>OVERLAY</td>
<td>UESCAPE</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>PERCENT_RANK</td>
<td>UNKNOWN</td>
</tr>
<tr>
<td>DEREF</td>
<td>PERCENTILE_CONT</td>
<td>UNNEST</td>
</tr>
<tr>
<td>ELEMENT</td>
<td>PERCENTILE_DISC</td>
<td>UPPER</td>
</tr>
<tr>
<td>EXEC</td>
<td>POWER</td>
<td>VAR_POP</td>
</tr>
<tr>
<td>EXP</td>
<td>REAL</td>
<td>VAR_SAMP</td>
</tr>
<tr>
<td>FALSE</td>
<td>RECURSIVE</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>FILTER</td>
<td>REF</td>
<td>VARYING</td>
</tr>
</tbody>
</table>
Reserved schema names and reserved words

<table>
<thead>
<tr>
<th>FLOAT</th>
<th>REGR_AVGX</th>
<th>WIDTH_BUCKET</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLOOR</td>
<td>REGR_AVGY</td>
<td>WINDOW</td>
</tr>
<tr>
<td>FUSION</td>
<td>REGR_COUNT</td>
<td>WITHIN</td>
</tr>
</tbody>
</table>
Appendix H. Examples of interaction between triggers and referential constraints

Update operations can cause the interaction of triggers with referential constraints and check constraints.

Figure 18 and the associated description are representative of the processing that is performed for an statement that updates data in the database.

Figure 18 shows the general order of processing for an statement that updates a table. It assumes a situation where the table includes BEFORE triggers, referential constraints, check constraints and AFTER triggers that cascade. The following is a description of the boxes and other items found in Figure 18:

- **statement S₁**
  This is the DELETE, INSERT, or UPDATE statement that begins the process. The statement $S₁$ identifies a table (or an updatable view over some table) referred to as the subject table throughout this description.

- **Determine set of affected rows**
  This step is the starting point for a process that repeats for referential constraint delete rules of CASCADE and SET NULL and for cascaded statements from AFTER triggers.

The purpose of this step is to determine the set of affected rows for the statement. The set of rows included is based on the statement:

- for DELETE, all rows that satisfy the search condition of the statement (or the current row for a positioned DELETE)
- for INSERT, the rows identified by the VALUES clause or the fullselect
- for UPDATE, all rows that satisfy the search condition (or the current row for a positioned UPDATE).
Examples of interaction between triggers and referential constraints

If the set of affected rows is empty, there will be no BEFORE triggers, changes to apply to the subject table, or constraints to process for the statement.

- **Process BEFORE triggers**
  All BEFORE triggers are processed in ascending order of creation. Each BEFORE trigger will process the triggered action once for each row in the set of affected rows.
  An error can occur during the processing of a triggered action in which case all changes made as a result of the original statement $S_1$ (so far) are rolled back.
  If there are no BEFORE triggers or the set of affected is empty, this step is skipped.

- **Apply the set of affected rows to the subject table**
  The actual delete, insert, or update is applied using the set of affected rows to the subject table in the database.
  An error can occur when applying the set of affected rows (such as attempting to insert a row with a duplicate key where a unique index exists) in which case all changes made as a result of the original statement $S_1$ (so far) are rolled back.

- **Apply Constraints**
  The constraints associated with the subject table are applied if set of affected rows is not empty. This includes unique constraints, unique indexes, referential constraints, check constraints and checks related to the WITH CHECK OPTION on views. Referential constraints with delete rules of cascade or set null might cause additional triggers to be activated.
  A violation of any constraint or WITH CHECK OPTION results in an error and all changes made as a result of $S_1$ (so far) are rolled back.

- **Process AFTER triggers**
  All AFTER triggers activated by $S_1$ are processed in ascending order of creation.
  FOR EACH STATEMENT triggers will process the triggered action exactly once, even if the set of affected rows is empty. FOR EACH ROW triggers will process the triggered action once for each row in the set of affected rows.
  An error can occur during the processing of a triggered action in which case all changes made as a result of the original $S_1$ (so far) are rolled back.
  The triggered action of a trigger can include triggered statements that are DELETE, INSERT or UPDATE statements. For the purposes of this description, each such statement is considered a cascaded statement.
  A cascaded statement is a DELETE, INSERT, or UPDATE statement that is processed as part of the triggered action of an AFTER trigger. This statement starts a cascaded level of trigger processing. This can be thought of as assigning the triggered statement as a new $S_1$ and performing all of the steps described here recursively.
  Once all triggered statements from all AFTER triggers activated by each $S_1$ have been processed to completion, the processing of the original $S_1$ is completed.

- **R = roll back changes to before $S_1$**
  Any error (including constraint violations) that occurs during processing results in a roll back of all the changes made directly or indirectly as a result of the original statement $S_1$. The database is therefore back in the same state as immediately prior to the execution of the original statement $S_1$. 

Appendix I. Explain tables

The Explain tables capture access plans when the Explain facility is activated. The Explain tables must be created before Explain can be invoked. You can create them using one of the following methods:

- Call the SYSPROC.SYSINSTALLOBJECTS procedure:
  ```sql
db2 CONNECT TO database-name
  db2 CALL SYSPROC.SYSINSTALLOBJECTS('EXPLAIN', 'C',
  CAST(NULL AS VARCHAR(128)), CAST(NULL AS VARCHAR(128))
  
  This call creates the explain tables under the SYSTOOLS schema. To create them under a different schema, specify a schema name as the last parameter in the call.
- Run the EXPLAIN.DDL DB2 command file:
  ```sql
db2 CONNECT TO database-name
  db2 -tf EXPLAIN.DDL
  
  This command file creates explain tables under the current schema. It is located at the DB2PATH\misc directory on Windows operating systems, and the INSTHOME/sqllib/misc directory on Linux and UNIX operating systems. DB2PATH is the location where you install your DB2 copy and INSTHOME is the instance home directory.

The Explain facility uses the following IDs as the schema when qualifying Explain tables that it is populating:

- The session authorization ID for dynamic SQL
- The statement authorization ID for static SQL

The schema can be associated with a set of Explain tables, or aliases that point to a set of Explain tables under a different schema. If no Explain tables are found under the schema, the Explain facility checks for Explain tables under the SYSTOOLS schema and attempts to use those tables.

The population of the Explain tables by the Explain facility will not activate triggers or referential or check constraints. For example, if an insert trigger were defined on the EXPLAIN_INSTANCE table, and an eligible statement were explained, the trigger would not be activated.

To improve the performance of the Explain facility in a partitioned database system, it is recommended that the Explain tables be created in a single partition database partition group, preferably on the same database partition to which you will be connected when compiling the query.
## ADVISE_INDEX Table

The ADVISE_INDEX table represents the recommended indexes.

### Table 238. ADVISE_INDEX Table

PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable?</th>
<th>Key?</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXPLAIN_REQUESTER</td>
<td>VARCHAR(128)</td>
<td>No</td>
<td>No</td>
<td>Authorization ID of initiator of this Explain request.</td>
</tr>
<tr>
<td>EXPLAIN_TIME</td>
<td>TIMESTAMP</td>
<td>No</td>
<td>No</td>
<td>Time of initiation for Explain request.</td>
</tr>
<tr>
<td>SOURCE_NAME</td>
<td>VARCHAR(128)</td>
<td>No</td>
<td>No</td>
<td>Name of the package running when the dynamic statement was explained or name of the source file when static SQL was explained.</td>
</tr>
<tr>
<td>SOURCE_SCHEMA</td>
<td>VARCHAR(128)</td>
<td>No</td>
<td>No</td>
<td>Schema, or qualifier, of source of Explain request.</td>
</tr>
<tr>
<td>SOURCE_VERSION</td>
<td>VARCHAR(64)</td>
<td>No</td>
<td>No</td>
<td>Version of the source of the Explain request.</td>
</tr>
<tr>
<td>EXPLAIN_LEVEL</td>
<td>CHAR(1)</td>
<td>No</td>
<td>No</td>
<td>Level of Explain information for which this row is relevant.</td>
</tr>
<tr>
<td>STMTNO</td>
<td>INTEGER</td>
<td>No</td>
<td>No</td>
<td>Statement number within package to which this explain information is related.</td>
</tr>
<tr>
<td>SECTNO</td>
<td>INTEGER</td>
<td>No</td>
<td>No</td>
<td>Section number within package to which this explain information is related.</td>
</tr>
<tr>
<td>QUERYNO</td>
<td>INTEGER</td>
<td>No</td>
<td>No</td>
<td>Numeric identifier for explained SQL statement. For dynamic SQL statements (excluding the EXPLAIN SQL statement) issued through CLP or CLI, the default value is a sequentially incremented value. Otherwise, the default value is the value of STMTNO for static SQL statements and 1 for dynamic SQL statements.</td>
</tr>
<tr>
<td>QUERYTAG</td>
<td>CHAR(20)</td>
<td>No</td>
<td>No</td>
<td>Identifier tag for each explained SQL statement. For dynamic SQL statements issued through CLP (excluding the EXPLAIN SQL statement), the default value is ‘CLP’. For dynamic SQL statements issued through CLI (excluding the EXPLAIN SQL statement), the default value is ‘CLI’. Otherwise, the default value used is blanks.</td>
</tr>
<tr>
<td>NAME</td>
<td>VARCHAR(128)</td>
<td>No</td>
<td>No</td>
<td>Name of the index.</td>
</tr>
<tr>
<td>CREATOR</td>
<td>VARCHAR(128)</td>
<td>No</td>
<td>No</td>
<td>Qualifier of the index name.</td>
</tr>
<tr>
<td>TBNNAME</td>
<td>VARCHAR(128)</td>
<td>No</td>
<td>No</td>
<td>Name of the table or nickname on which the index is defined.</td>
</tr>
<tr>
<td>TBCREATOR</td>
<td>VARCHAR(128)</td>
<td>No</td>
<td>No</td>
<td>Qualifier of the table name.</td>
</tr>
<tr>
<td>COLNAMES</td>
<td>CLOB(2M)</td>
<td>No</td>
<td>No</td>
<td>List of column names.</td>
</tr>
<tr>
<td>UNIQUERULE</td>
<td>CHAR(1)</td>
<td>No</td>
<td>No</td>
<td>Unique rule: • D = Duplicates allowed • P = Primary index • U = Unique entries only allowed</td>
</tr>
<tr>
<td>COLCOUNT</td>
<td>SMALLINT</td>
<td>No</td>
<td>No</td>
<td>Number of columns in the key plus the number of include columns if any.</td>
</tr>
<tr>
<td>IID</td>
<td>SMALLINT</td>
<td>No</td>
<td>No</td>
<td>Internal index ID.</td>
</tr>
<tr>
<td>NLEAF</td>
<td>BIGINT</td>
<td>No</td>
<td>No</td>
<td>Number of leaf pages; -1 if statistics are not gathered.</td>
</tr>
</tbody>
</table>
**ADVISE_INDEX table**

Table 238. ADVISE_INDEX Table (continued). PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable?</th>
<th>Key?</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NLEVELS</td>
<td>SMALLINT</td>
<td>No</td>
<td>No</td>
<td>Number of index levels; -1 if statistics are not gathered.</td>
</tr>
<tr>
<td>FIRSTKEYCARD</td>
<td>BIGINT</td>
<td>No</td>
<td>No</td>
<td>Number of distinct first key values; -1 if statistics are not gathered.</td>
</tr>
<tr>
<td>FULLKEYCARD</td>
<td>BIGINT</td>
<td>No</td>
<td>No</td>
<td>Number of distinct full key values; -1 if statistics are not gathered.</td>
</tr>
<tr>
<td>CLUSTERRATIO</td>
<td>SMALLINT</td>
<td>No</td>
<td>No</td>
<td>Degree of data clustering with the index; -1 if statistics are not gathered or if detailed index statistics are gathered (in which case CLUSTERFACTOR will be used instead).</td>
</tr>
<tr>
<td>AVGPARTITION_</td>
<td>SMALLINT</td>
<td>No</td>
<td>No</td>
<td>Degree of data clustering within a single data partition. -1 if the table is not table partitioned, if statistics are not gathered, or if detailed statistics are gathered (in which case AVGPARTITION_CLUSTERFACTOR will be used instead).</td>
</tr>
<tr>
<td>CLUSTERRATIO</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AVGPARTITION_</td>
<td>DOUBLE</td>
<td>No</td>
<td>No</td>
<td>Finer measurement of the degree of clustering within a single data partition. -1 if the table is not table partitioned, if statistics are not gathered, or if the index is defined on a nickname.</td>
</tr>
<tr>
<td>CLUSTERFACTOR</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>USERDEFINED</td>
<td>SMALLINT</td>
<td>No</td>
<td>No</td>
<td>Defined by the user.</td>
</tr>
</tbody>
</table>

Appendix I. Explain tables 1039
**ADVISE_INDEX table**

*Table 238. ADVISE_INDEX Table (continued).* PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable?</th>
<th>Key?</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYSTEM_REQUIRED</td>
<td>SMALLINT</td>
<td>No</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1 if one or the other of the following conditions is met:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>– This index is required for a primary or unique key constraint, or this index is a dimension block index or composite block index for a multi-dimensional clustering (MDC) table.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>– This is an index on the (OID) column of a typed table.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2 if both of the following conditions are met:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>– This index is required for a primary or unique key constraint, or this index is a dimension block index or composite block index for an MDC table.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>– This is an index on the (OID) column of a typed table.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0 otherwise.</td>
</tr>
<tr>
<td>CREATE_TIME</td>
<td>TIMESTAMP</td>
<td>No</td>
<td>No</td>
<td>Time when the index was created.</td>
</tr>
<tr>
<td>STATS_TIME</td>
<td>TIMESTAMP</td>
<td>Yes</td>
<td>No</td>
<td>Last time when any change was made to recorded statistics for this index. Null if no statistics available.</td>
</tr>
<tr>
<td>PAGE_FETCH_PAIRS</td>
<td>VARCHAR(520)</td>
<td>No</td>
<td>No</td>
<td>A list of pairs of integers, represented in character form. Each pair represents the number of pages in a hypothetical buffer, and the number of page fetches required to scan the table with this index using that hypothetical buffer. (Zero-length string if no data available.)</td>
</tr>
<tr>
<td>REMARKS</td>
<td>VARCHAR(254)</td>
<td>Yes</td>
<td>No</td>
<td>User-supplied comment, or null.</td>
</tr>
<tr>
<td>DEFINER</td>
<td>VARCHAR(128)</td>
<td>No</td>
<td>No</td>
<td>User who created the index.</td>
</tr>
<tr>
<td>CONVERTED</td>
<td>CHAR(1)</td>
<td>No</td>
<td>No</td>
<td>Reserved for future use.</td>
</tr>
<tr>
<td>SEQUENTIAL_PAGES</td>
<td>BIGINT</td>
<td>No</td>
<td>No</td>
<td>Number of leaf pages located on disk in index key order with few or no large gaps between them. (-1 if no statistics are available.)</td>
</tr>
<tr>
<td>DENSITY</td>
<td>INTEGER</td>
<td>No</td>
<td>No</td>
<td>Ratio of SEQUENTIAL_PAGES to number of pages in the range of pages occupied by the index, expressed as a percent (integer between 0 and 100, -1 if no statistics are available.)</td>
</tr>
<tr>
<td>FIRST2KEYCARD</td>
<td>BIGINT</td>
<td>No</td>
<td>No</td>
<td>Number of distinct keys using the first two columns of the index (-1 if no statistics or inapplicable)</td>
</tr>
<tr>
<td>FIRST3KEYCARD</td>
<td>BIGINT</td>
<td>No</td>
<td>No</td>
<td>Number of distinct keys using the first three columns of the index (-1 if no statistics or inapplicable)</td>
</tr>
<tr>
<td>FIRST4KEYCARD</td>
<td>BIGINT</td>
<td>No</td>
<td>No</td>
<td>Number of distinct keys using the first four columns of the index (-1 if no statistics or inapplicable)</td>
</tr>
<tr>
<td>PCTFREE</td>
<td>SMALLINT</td>
<td>No</td>
<td>No</td>
<td>Percentage of each index leaf page to be reserved during initial building of the index. This space is available for future inserts after the index is built.</td>
</tr>
</tbody>
</table>
### ADVISE_INDEX table

*Table 238. ADVISE_INDEX Table (continued).* PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable?</th>
<th>Key?</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNIQUE_COLCOUNT</td>
<td>SMALLINT</td>
<td>No</td>
<td>No</td>
<td>The number of columns required for a unique key. Always &lt;=COLCOUNT. &lt; COLCOUNT only if there are include columns. -1 if index has no unique key (permits duplicates)</td>
</tr>
<tr>
<td>MINPCTUSED</td>
<td>SMALLINT</td>
<td>No</td>
<td>No</td>
<td>If not zero, then online index defragmentation is enabled, and the value is the threshold of minimum used space before merging pages.</td>
</tr>
</tbody>
</table>
| REVERSE_SCANS         | CHAR(1)       | No        | No   | • Y = Index supports reverse scans  
                          • N = Index does not support reverse scans                                    |
| USE_INDEX             | CHAR(1)       | Yes       | No   | • Y = index recommended or evaluated  
                          • N = index not to be recommended  
                          • R = an existing clustering RID index was recommended (by the Design Advisor) to be unclustered; this is the case when a new clustering RID index is recommended for the table |
| CREATION_TEXT         | CLOB(2M)      | No        | No   | The SQL statement used to create the index.                                                                                               |
| PACKED_DESC           | BLOB(1M)      | Yes       | No   | Internal description of the table.                                                                                                         |
| RUN_ID                | TIMESTAMP     | Yes       | FK   | A value corresponding to the START_TIME of a row in the ADVISE_INSTANCE table, linking it to the same Design Advisor run.                   |
| INDEXTYPE             | VARCHAR(4)    | No        | No   | Type of index.  
                          • CLUS = Clustering  
                          • REG = Regular  
                          • DIM = Dimension block index  
                          • BLOK = Block index                                                                 |
| EXISTS                | CHAR(1)       | No        | No   | Set to ‘Y’ if the index exists in the database catalog.                                                                                     |
| RIDTOBLOCK            | CHAR(1)       | No        | No   | Set to ‘Y’ if the RID index was used to make a block index in the Design Advisor.                                                           |
The ADVISE_INSTANCE table contains information about db2advis execution, including start time. Contains one row for each execution of db2advis. Other ADVISE tables have a foreign key (RUN_ID) that links to the START_TIME column of the ADVISE_INSTANCE table for rows created during the same Design Advisor run.

### Table 239. ADVISE_INSTANCE Table

PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
<th>Key?</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>START_TIME</td>
<td>TIMESTAMP</td>
<td>No</td>
<td>PK</td>
<td>Time at which db2advis execution begins.</td>
</tr>
<tr>
<td>END_TIME</td>
<td>TIMESTAMP</td>
<td>No</td>
<td>No</td>
<td>Time at which db2advis execution ends.</td>
</tr>
<tr>
<td>MODE</td>
<td>VARCHAR(4)</td>
<td>No</td>
<td>No</td>
<td>The value that was specified with the -m option on the Design Advisor; for example, 'MC' to specify MQT and MDC.</td>
</tr>
<tr>
<td>WKLD_COMPRESSION</td>
<td>CHAR(4)</td>
<td>No</td>
<td>No</td>
<td>The workload compression under which the Design Advisor was run.</td>
</tr>
<tr>
<td>STATUS</td>
<td>CHAR(9)</td>
<td>No</td>
<td>No</td>
<td>The status of a Design Advisor run. Status can be 'STARTED', 'COMPLETED' (if successful), or an error number that is prefixed by 'EI' for internal errors or 'EX' for external errors, in which case the error number represents the SQLCODE.</td>
</tr>
</tbody>
</table>
The ADVISE_MQT table contains information about materialized query tables (MQT) recommended by the Design Advisor.

Table 240. ADVISE_MQT Table. PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable?</th>
<th>Key?</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXPLAIN_REQUESTER</td>
<td>VARCHAR(128)</td>
<td>No</td>
<td>No</td>
<td>Authorization ID of initiator of this Explain request.</td>
</tr>
<tr>
<td>EXPLAIN_TIME</td>
<td>TIMESTAMP</td>
<td>No</td>
<td>No</td>
<td>Time of initiation for Explain request.</td>
</tr>
<tr>
<td>SOURCE_NAME</td>
<td>VARCHAR(128)</td>
<td>No</td>
<td>No</td>
<td>Name of the package running when the dynamic statement was explained or name of the source file when the static SQL was explained.</td>
</tr>
<tr>
<td>SOURCE_SCHEMA</td>
<td>VARCHAR(128)</td>
<td>No</td>
<td>No</td>
<td>Schema, or qualifier, of source of Explain request.</td>
</tr>
<tr>
<td>SOURCE_VERSION</td>
<td>VARCHAR(64)</td>
<td>No</td>
<td>No</td>
<td>Version of the source of the Explain request.</td>
</tr>
<tr>
<td>EXPLAIN_LEVEL</td>
<td>CHAR(1)</td>
<td>No</td>
<td>No</td>
<td>Level of Explain information for which this row is relevant.</td>
</tr>
<tr>
<td>STMTNO</td>
<td>INTEGER</td>
<td>No</td>
<td>No</td>
<td>Statement number within package to which this Explain information is related.</td>
</tr>
<tr>
<td>SECTNO</td>
<td>INTEGER</td>
<td>No</td>
<td>No</td>
<td>Statement number within package to which this Explain information is related.</td>
</tr>
<tr>
<td>NAME</td>
<td>VARCHAR(128)</td>
<td>No</td>
<td>No</td>
<td>MQT name.</td>
</tr>
<tr>
<td>CREATOR</td>
<td>VARCHAR(128)</td>
<td>No</td>
<td>No</td>
<td>MQT creator name.</td>
</tr>
<tr>
<td>IID</td>
<td>SMALLINT</td>
<td>No</td>
<td>No</td>
<td>Internal identifier.</td>
</tr>
<tr>
<td>CREATE_TIME</td>
<td>TIMESTAMP</td>
<td>No</td>
<td>No</td>
<td>Time at which the MQT was created.</td>
</tr>
<tr>
<td>STATS_TIME</td>
<td>TIMESTAMP</td>
<td>Yes</td>
<td>No</td>
<td>Time at which statistics were taken.</td>
</tr>
<tr>
<td>NUMROWS</td>
<td>DOUBLE</td>
<td>No</td>
<td>No</td>
<td>The number of estimated rows in the MQT.</td>
</tr>
<tr>
<td>NUMCOLS</td>
<td>SMALLINT</td>
<td>No</td>
<td>No</td>
<td>Number of columns defined in the MQT.</td>
</tr>
<tr>
<td>ROWSIZE</td>
<td>DOUBLE</td>
<td>No</td>
<td>No</td>
<td>Average length (in bytes) of a row in the MQT.</td>
</tr>
<tr>
<td>BENEFIT</td>
<td>FLOAT</td>
<td>No</td>
<td>No</td>
<td>Reserved for future use.</td>
</tr>
<tr>
<td>USE_MQT</td>
<td>CHAR(1)</td>
<td>Yes</td>
<td>No</td>
<td>Set to 'Y' when the MQT is recommended.</td>
</tr>
<tr>
<td>MQT_SOURCE</td>
<td>CHAR(1)</td>
<td>Yes</td>
<td>No</td>
<td>Indicates where the MQT candidate was generated. Set to 'I' if the MQT candidate is a refresh-immediate MQT, or 'D' if it can only be created as a full refresh-deferred MQT.</td>
</tr>
<tr>
<td>QUERY_TEXT</td>
<td>CLOB(2M)</td>
<td>No</td>
<td>No</td>
<td>Contains the query that defines the MQT.</td>
</tr>
<tr>
<td>CREATION_TEXT</td>
<td>CLOB(2M)</td>
<td>No</td>
<td>No</td>
<td>Contains the CREATE TABLE DDL for the MQT.</td>
</tr>
<tr>
<td>SAMPLE_TEXT</td>
<td>CLOB(2M)</td>
<td>No</td>
<td>No</td>
<td>Contains the sampling query that is used to get detailed statistics for the MQT. Only used when detailed statistics are required for the Design Advisor. The resulting sampled statistics will be shown in this table. If null, then no sampling query was created for this MQT.</td>
</tr>
<tr>
<td>COLSTATS</td>
<td>CLOB(2M)</td>
<td>No</td>
<td>No</td>
<td>Contains the column statistics for the MQT (if not null). These statistics are in XML format and include the column name, column cardinality and, optionally, the HIGH2KEY and LOW2KEY values.</td>
</tr>
</tbody>
</table>
### ADVISE_MQT table

*Table 240. ADVISE_MQT Table (continued).* PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable?</th>
<th>Key?</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXTRA_INFO</td>
<td>BLOB(2M)</td>
<td>No</td>
<td>No</td>
<td>Reserved for miscellaneous output.</td>
</tr>
<tr>
<td>TBSPACE</td>
<td>VARCHAR(128)</td>
<td>No</td>
<td>No</td>
<td>The table space that is recommended for the MQT.</td>
</tr>
<tr>
<td>RUN_ID</td>
<td>TIMESTAMP</td>
<td>Yes</td>
<td>FK</td>
<td>A value corresponding to the START_TIME of a row in the ADVISE_INSTANCE table, linking it to the same Design Advisor run.</td>
</tr>
<tr>
<td>REFRESH_TYPE</td>
<td>CHAR(1)</td>
<td>No</td>
<td>No</td>
<td>Set to ‘I’ for immediate or ‘D’ for deferred.</td>
</tr>
<tr>
<td>EXISTS</td>
<td>CHAR(1)</td>
<td>No</td>
<td>No</td>
<td>Set to ‘Y’ if the MQT exists in the database catalog.</td>
</tr>
</tbody>
</table>
The ADVISE_PARTITION table contains information about database partitions recommended by the Design Advisor, and can only be populated in a partitioned database environment.

Table 241. ADVISE_PARTITION Table. PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable?</th>
<th>Key?</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXPLAIN_REQUESTER</td>
<td>VARCHAR(128)</td>
<td>No</td>
<td>No</td>
<td>Authorization ID of initiator of this Explain request.</td>
</tr>
<tr>
<td>EXPLAIN_TIME</td>
<td>TIMESTAMP</td>
<td>No</td>
<td>No</td>
<td>Time of initiation for Explain request.</td>
</tr>
<tr>
<td>SOURCE_NAME</td>
<td>VARCHAR(128)</td>
<td>No</td>
<td>No</td>
<td>Name of the package running when the dynamic statement was explained or name of the source file when the static SQL was explained.</td>
</tr>
<tr>
<td>SOURCE_SCHEMA</td>
<td>VARCHAR(128)</td>
<td>No</td>
<td>No</td>
<td>Schema, or qualifier, of source of Explain request.</td>
</tr>
<tr>
<td>SOURCE_VERSION</td>
<td>VARCHAR(64)</td>
<td>No</td>
<td>No</td>
<td>Version of the source of the Explain request.</td>
</tr>
<tr>
<td>EXPLAIN_LEVEL</td>
<td>CHAR(1)</td>
<td>No</td>
<td>No</td>
<td>Level of Explain information for which this row is relevant.</td>
</tr>
<tr>
<td>STMTNO</td>
<td>INTEGER</td>
<td>No</td>
<td>No</td>
<td>Statement number within package to which this Explain information is related.</td>
</tr>
<tr>
<td>SECTNO</td>
<td>INTEGER</td>
<td>No</td>
<td>No</td>
<td>Statement number within package to which this Explain information is related.</td>
</tr>
<tr>
<td>QUERYNO</td>
<td>INTEGER</td>
<td>No</td>
<td>No</td>
<td>Numeric identifier for explained SQL statement. For dynamic SQL statements (excluding the EXPLAIN SQL statement) issued through CLP or CLI, the default value is a sequentially incremented value. Otherwise, the default value is the value of STMTNO for static SQL statements and 1 for dynamic SQL statements.</td>
</tr>
<tr>
<td>QUERYTAG</td>
<td>CHAR(20)</td>
<td>No</td>
<td>No</td>
<td>Identifier tag for each explained SQL statement. For dynamic SQL statements issued through CLP (excluding the EXPLAIN SQL statement), the default value is 'CLP'. For dynamic SQL statements issued through CLI (excluding the EXPLAIN SQL statement), the default value is 'CLI'. Otherwise, the default value used is blanks.</td>
</tr>
<tr>
<td>TBNAME</td>
<td>VARCHAR(128)</td>
<td>Yes</td>
<td>No</td>
<td>Specifies the table name.</td>
</tr>
<tr>
<td>TBCREATOR</td>
<td>VARCHAR(128)</td>
<td>Yes</td>
<td>No</td>
<td>Specifies the table creator name.</td>
</tr>
<tr>
<td>PMID</td>
<td>SMALLINT</td>
<td>Yes</td>
<td>No</td>
<td>Specifies the distribution map ID.</td>
</tr>
<tr>
<td>TBSPACE</td>
<td>VARCHAR(128)</td>
<td>Yes</td>
<td>No</td>
<td>Specifies the table space in which the table resides.</td>
</tr>
<tr>
<td>COLNAMES</td>
<td>CLOB(2M)</td>
<td>Yes</td>
<td>No</td>
<td>Specifies database partition column names, separated by commas.</td>
</tr>
<tr>
<td>COLCOUNT</td>
<td>SMALLINT</td>
<td>Yes</td>
<td>No</td>
<td>Specifies the number of database partitioning columns.</td>
</tr>
<tr>
<td>REPLICATE</td>
<td>CHAR(1)</td>
<td>Yes</td>
<td>No</td>
<td>Specifies whether or not the database partition is replicated.</td>
</tr>
<tr>
<td>COST</td>
<td>DOUBLE</td>
<td>Yes</td>
<td>No</td>
<td>Specifies the cost of using the database partition.</td>
</tr>
</tbody>
</table>
**ADVISE_PARTITION table**

*Table 241. ADVISE_PARTITION Table (continued).* PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable?</th>
<th>Key?</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>USEIT</td>
<td>CHAR(1)</td>
<td>Yes</td>
<td>No</td>
<td>Specifies whether or not the database partition is used in EVALUATE PARTITION mode. A database partition is used if USEIT is set to 'Y' or 'y'.</td>
</tr>
<tr>
<td>RUN_ID</td>
<td>TIMESTAMP</td>
<td>Yes</td>
<td>FK</td>
<td>A value corresponding to the START_TIME of a row in the ADVISE_INSTANCE table, linking it to the same Design Advisor run.</td>
</tr>
</tbody>
</table>
**ADVISE_TABLE table**

The ADVISE_TABLE table stores the data definition language (DDL) for table creation, using the final Design Advisor recommendations for materialized query tables (MQTs), multidimensional clustered tables (MDCs), and database partitioning.

*Table 242. ADVISE_TABLE Table.* PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable?</th>
<th>Key?</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RUN_ID</td>
<td>TIMESTAMP</td>
<td>Yes</td>
<td>FK</td>
<td>A value corresponding to the START_TIME of a row in the ADVISE_INSTANCE table, linking it to the same Design Advisor run.</td>
</tr>
<tr>
<td>TABLE_NAME</td>
<td>VARCHAR(128)</td>
<td>No</td>
<td>No</td>
<td>Name of the table.</td>
</tr>
<tr>
<td>TABLE_SCHEMA</td>
<td>VARCHAR(128)</td>
<td>No</td>
<td>No</td>
<td>Name of the table creator.</td>
</tr>
<tr>
<td>TABLESPACE</td>
<td>VARCHAR(128)</td>
<td>No</td>
<td>No</td>
<td>The table space in which the table is to be created.</td>
</tr>
<tr>
<td>SELECTION_FLAG</td>
<td>VARCHAR(4)</td>
<td>No</td>
<td>No</td>
<td>Indicates the recommendation type. Valid values are 'M' for MQT, 'P' for database partitioning, and 'C' for MDC. This field can include any subset of these values. For example, 'MC' indicates that the table is recommended as an MQT and an MDC table.</td>
</tr>
<tr>
<td>TABLE_EXISTS</td>
<td>CHAR(1)</td>
<td>No</td>
<td>No</td>
<td>Set to 'Y' if the table exists in the database catalog.</td>
</tr>
<tr>
<td>USE_TABLE</td>
<td>CHAR(1)</td>
<td>No</td>
<td>No</td>
<td>Set to 'Y' if the table has recommendations from the Design Advisor.</td>
</tr>
<tr>
<td>GEN_COLUMNS</td>
<td>CLOB(2M)</td>
<td>No</td>
<td>No</td>
<td>Contains a generated columns string if this row includes an MDC recommendation that requires generated columns in the create table DDL.</td>
</tr>
<tr>
<td>ORGANIZE_BY</td>
<td>CLOB(2M)</td>
<td>No</td>
<td>No</td>
<td>For MDC recommendations, contains the ORGANIZE BY clause of the create table DDL.</td>
</tr>
<tr>
<td>CREATION_TEXT</td>
<td>CLOB(2M)</td>
<td>No</td>
<td>No</td>
<td>Contains the create table DDL.</td>
</tr>
<tr>
<td>ALTER_COMMAND</td>
<td>CLOB(2M)</td>
<td>No</td>
<td>No</td>
<td>Contains an ALTER TABLE statement for the table.</td>
</tr>
</tbody>
</table>
ADVISE_WORKLOAD table

The ADVISE_WORKLOAD table represents the statement that makes up the workload.

Table 243. ADVISE_WORKLOAD Table. PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable?</th>
<th>Key?</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WORKLOAD_NAME</td>
<td>CHAR(128)</td>
<td>No</td>
<td>No</td>
<td>Name of the collection of SQL statements (workload) to which this statement belongs.</td>
</tr>
<tr>
<td>STATEMENT_NO</td>
<td>INTEGER</td>
<td>No</td>
<td>No</td>
<td>Statement number within the workload to which this explain information is related.</td>
</tr>
<tr>
<td>STATEMENT_TEXT</td>
<td>CLOB(1M)</td>
<td>No</td>
<td>No</td>
<td>Content of the SQL statement.</td>
</tr>
<tr>
<td>STATEMENT_TAG</td>
<td>VARCHAR(256)</td>
<td>No</td>
<td>No</td>
<td>Identifier tag for each explained SQL statement.</td>
</tr>
<tr>
<td>FREQUENCY</td>
<td>INTEGER</td>
<td>No</td>
<td>No</td>
<td>The number of times this statement appears within the workload.</td>
</tr>
<tr>
<td>IMPORTANCE</td>
<td>DOUBLE</td>
<td>No</td>
<td>No</td>
<td>Importance of the statement.</td>
</tr>
<tr>
<td>WEIGHT</td>
<td>DOUBLE</td>
<td>No</td>
<td>No</td>
<td>Priority of the statement.</td>
</tr>
<tr>
<td>COST_BEFORE</td>
<td>DOUBLE</td>
<td>Yes</td>
<td>No</td>
<td>The cost of the query (in timerons) if the recommendations are not created.</td>
</tr>
<tr>
<td>COST_AFTER</td>
<td>DOUBLE</td>
<td>Yes</td>
<td>No</td>
<td>The cost of the query (in timerons) if the recommendations are created.</td>
</tr>
<tr>
<td>COMPILABLE</td>
<td>CHAR(17)</td>
<td>Yes</td>
<td>No</td>
<td>Indicates any query compile errors that occurred while trying to prepare the statement. If this column is NULL or does not start with SQLCA, the SQL query could be compiled by db2advis. If a compile error is found by db2advis or the Design Advisor, the COMPILABLE column value consists of an 8 byte long SQLCA.sqlcaid field, followed by a colon (:) and an 8 byte long SQLCA.sqlstate field, which is the return code for the SQL statement.</td>
</tr>
</tbody>
</table>
The EXPLAIN_ACTUALS table contains Explain section actuals information.

Table 244. EXPLAIN_ACTUALS Table. PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable?</th>
<th>Key?</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXPLAINREQUESTER</td>
<td>VARCHAR(128)</td>
<td>No</td>
<td>FK</td>
<td>Authorization ID of initiator of this Explain request.</td>
</tr>
<tr>
<td>EXPLAIN_TIME</td>
<td>TIMESTAMP</td>
<td>No</td>
<td>FK</td>
<td>Time of initiation for Explain request.</td>
</tr>
<tr>
<td>SOURCE_NAME</td>
<td>VARCHAR(128)</td>
<td>No</td>
<td>FK</td>
<td>Name of the package running when the dynamic statement was explained or name of the source file when static SQL was explained.</td>
</tr>
<tr>
<td>SOURCE_SCHEMA</td>
<td>VARCHAR(128)</td>
<td>No</td>
<td>FK</td>
<td>Schema, or qualifier, of source of Explain request.</td>
</tr>
<tr>
<td>SOURCE_VERSION</td>
<td>VARCHAR(64)</td>
<td>No</td>
<td>FK</td>
<td>Version of the source of the Explain request.</td>
</tr>
<tr>
<td>EXPLAIN_LEVEL</td>
<td>CHAR(1)</td>
<td>No</td>
<td>FK</td>
<td>Level of Explain information for which this row is relevant.</td>
</tr>
<tr>
<td>STMTNO</td>
<td>INTEGER</td>
<td>No</td>
<td>FK</td>
<td>Statement number within package to which this Explain information is related.</td>
</tr>
<tr>
<td>SECTNO</td>
<td>INTEGER</td>
<td>No</td>
<td>FK</td>
<td>Section number within package to which this Explain information is related.</td>
</tr>
<tr>
<td>OPERATOR_ID</td>
<td>INTEGER</td>
<td>No</td>
<td>FK</td>
<td>Unique ID for this operator within this query.</td>
</tr>
<tr>
<td>DBPARTITIONNUM</td>
<td>INTEGER</td>
<td>No</td>
<td>No</td>
<td>The partition number of the database partition where the operator has run.</td>
</tr>
<tr>
<td>PREDICATE_ID</td>
<td>INTEGER</td>
<td>Yes</td>
<td>No</td>
<td>ID of the predicate applied on this operator. NULL if the actuals are operator actuals.</td>
</tr>
<tr>
<td>HOW_APPLIED</td>
<td>CHAR(10)</td>
<td>Yes</td>
<td>No</td>
<td>How predicate is used by this operator. NULL if PREDICATE_ID is NULL.</td>
</tr>
<tr>
<td>ACTUAL_TYPE</td>
<td>VARCHAR(12)</td>
<td>No</td>
<td>No</td>
<td>The type of the actual.</td>
</tr>
<tr>
<td>ACTUAL_VALUE</td>
<td>DOUBLE</td>
<td>Yes</td>
<td>No</td>
<td>The value of the actual. NULL if actual is not available for this operator.</td>
</tr>
</tbody>
</table>
The `EXPLAIN_ARGUMENT` table represents the unique characteristics for each individual operator, if there are any.

**Table 245. EXPLAIN_ARGUMENT Table.** PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable?</th>
<th>Key?</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXPLAIN_REQUESTER</td>
<td>VARCHAR(128)</td>
<td>No</td>
<td>FK</td>
<td>Authorization ID of initiator of this Explain request.</td>
</tr>
<tr>
<td>EXPLAIN_TIME</td>
<td>TIMESTAMP</td>
<td>No</td>
<td>FK</td>
<td>Time of initiation for Explain request.</td>
</tr>
<tr>
<td>SOURCE_NAME</td>
<td>VARCHAR(128)</td>
<td>No</td>
<td>FK</td>
<td>Name of the package running when the dynamic statement was explained or name of the source file when static SQL was explained.</td>
</tr>
<tr>
<td>SOURCE_SCHEMA</td>
<td>VARCHAR(128)</td>
<td>No</td>
<td>FK</td>
<td>Schema, or qualifier, of source of Explain request.</td>
</tr>
<tr>
<td>SOURCE_VERSION</td>
<td>VARCHAR(64)</td>
<td>No</td>
<td>FK</td>
<td>Version of the source of the Explain request.</td>
</tr>
<tr>
<td>EXPLAIN_LEVEL</td>
<td>CHAR(1)</td>
<td>No</td>
<td>FK</td>
<td>Level of Explain information for which this row is relevant.</td>
</tr>
<tr>
<td>STMTNO</td>
<td>INTEGER</td>
<td>No</td>
<td>FK</td>
<td>Statement number within package to which this Explain information is related.</td>
</tr>
<tr>
<td>SECTNO</td>
<td>INTEGER</td>
<td>No</td>
<td>FK</td>
<td>Section number within package to which this Explain information is related.</td>
</tr>
<tr>
<td>OPERATOR_ID</td>
<td>INTEGER</td>
<td>No</td>
<td>No</td>
<td>Unique ID for this operator within this query.</td>
</tr>
<tr>
<td>ARGUMENT_TYPE</td>
<td>CHAR(8)</td>
<td>No</td>
<td>No</td>
<td>The type of argument for this operator.</td>
</tr>
<tr>
<td>ARGUMENT_VALUE</td>
<td>VARCHAR(1024)</td>
<td>Yes</td>
<td>No</td>
<td>The value of the argument for this operator. NULL if the value is in LONG_ARGUMENT_VALUE.</td>
</tr>
<tr>
<td>LONG_ARGUMENT_VALUE</td>
<td>CLOB(2M)</td>
<td>Yes</td>
<td>No</td>
<td>The value of the argument for this operator, when the text will not fit in ARGUMENT_VALUE. NULL if the value is in ARGUMENT_VALUE.</td>
</tr>
</tbody>
</table>

**Table 246. ARGUMENT_TYPE and ARGUMENT_VALUE column values**

<table>
<thead>
<tr>
<th>ARGUMENT_TYPE Value</th>
<th>Possible ARGUMENT_VALUE Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AGGMODE</td>
<td>COMPLETE, PARTIAL, INTERMEDIATE, FINAL</td>
<td>Partial aggregation indicators.</td>
</tr>
<tr>
<td>BITFLTR</td>
<td>INTEGER, FALSE</td>
<td>Size of bit filter used by hash join.</td>
</tr>
<tr>
<td>BLD_LEVEL</td>
<td>DB2 Build Identifier</td>
<td>Internal identification string for source code version.</td>
</tr>
<tr>
<td>BLKLOCK</td>
<td>EXCLUSIVE, INTENT, EXCLUSIVE, INTENT, SHARE, NONE, SHARE, UPDATE</td>
<td>Block level lock intent.</td>
</tr>
</tbody>
</table>
### EXPLAIN_ARGUMENT table

Table 246. ARGUMENT_TYPE and ARGUMENT_VALUE column values  (continued)

<table>
<thead>
<tr>
<th>ARGUMENT_TYPE</th>
<th>Possible ARGUMENT_VALUE Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONCACCR</td>
<td>Each row of this type will contain:</td>
<td>Indicates the concurrent access resolution used to generate the access plan for this statement.</td>
</tr>
<tr>
<td></td>
<td>• Level of the setting for this statement:</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>BIND</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Application BIND with CONCURRENT ACCESS RESOLUTION option</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>PREP</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Statement prepared with CONCURRENT ACCESS RESOLUTION attributes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• The concurrent access resolution in effect:</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>USE CURRENTLY COMMITTED</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Concurrent access resolution of application bind or statement prepare is USE CURRENTLY COMMITTED</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>WAIT FOR OUTCOME</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Concurrent access resolution of application bind or statement prepare is WAIT FOR OUTCOME</td>
<td></td>
</tr>
<tr>
<td>CSERQY</td>
<td>TRUE</td>
<td>Remote query is a common subexpression.</td>
</tr>
<tr>
<td></td>
<td>FALSE</td>
<td></td>
</tr>
<tr>
<td>CSETEMP</td>
<td>TRUE</td>
<td>Temporary Table over Common Subexpression Flag.</td>
</tr>
<tr>
<td></td>
<td>FALSE</td>
<td></td>
</tr>
<tr>
<td>CUR_COMM</td>
<td>TRUE</td>
<td>Access currently committed rows when the value for the database configuration parameter <code>cur_commit</code> is not DISABLE. This access plan is enabled for applicable statements by using either:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• CONCURRENT ACCESS RESOLUTION with the USE CURRENTLY COMMITTED option on bind or prepare</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• The database configuration parameter <code>cur_commit</code> with a value of ON</td>
</tr>
<tr>
<td>DIRECT</td>
<td>TRUE</td>
<td>Direct fetch indicator.</td>
</tr>
<tr>
<td>DPESTFLG</td>
<td>TRUE</td>
<td>Indicates whether or not the DPNUMPRT value is based on an estimate. Possible values are ‘TRUE’ (DPNUMPRT represents the estimated number of accessed data partitions) or ‘FALSE’ (DPNUMPRT represents the actual number of accessed data partitions).</td>
</tr>
<tr>
<td></td>
<td>FALSE</td>
<td></td>
</tr>
<tr>
<td>DPLSTPRT</td>
<td>NONE</td>
<td>Represents accessed data partitions. It is a comma-delimited list (for example: 1,3,5) or a hyphenated list (for example: 1-5) of accessed data partitions. A value of ‘NONE’ means that no data partition remains after specified predicates have been applied.</td>
</tr>
<tr>
<td></td>
<td>CHARACTER</td>
<td></td>
</tr>
</tbody>
</table>
### EXPLAIN_ARGUMENT table

Table 246. ARGUMENT_TYPE and ARGUMENT_VALUE column values (continued)

<table>
<thead>
<tr>
<th>ARGUMENT_TYPE</th>
<th>Possible ARGUMENT_VALUE Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DPNUMPRT</td>
<td>INTEGER</td>
<td>Represents the actual or estimated number of data partitions accessed.</td>
</tr>
<tr>
<td>DSTSEVER</td>
<td>Server name</td>
<td>Destination (ship from) server.</td>
</tr>
<tr>
<td>DUPLWARN</td>
<td>TRUE, FALSE</td>
<td>Duplicates Warning flag.</td>
</tr>
<tr>
<td>EARLYOUT</td>
<td>LEFT, RIGHT, GROUPBY, NONE</td>
<td>Early out indicator. LEFT indicates that each row from the outer table only needs to be joined with at most one row from the inner table. RIGHT indicates that each row from the inner table only needs to be joined with at most one row from the outer table. NONE indicates no early out processing. GROUPBY indicates that early out processing is allowed because of a group by operation.</td>
</tr>
<tr>
<td>ENVVAR</td>
<td>Each row of this type will contain:</td>
<td>Environment variable affecting the optimizer</td>
</tr>
<tr>
<td></td>
<td>• Environment variable name</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Environment variable value</td>
<td></td>
</tr>
<tr>
<td>ERRTOL</td>
<td>Each row of this type will contain an SQLSTATE and SQLCODE pair.</td>
<td>A list of errors to be tolerated.</td>
</tr>
<tr>
<td>EVALUNCO</td>
<td>TRUE</td>
<td>Evaluate uncommitted data using lock deferral. This is enabled with the DB2_EVALUNCOMMITTED registry variable.</td>
</tr>
<tr>
<td>FETCHMAX</td>
<td>IGNORE, INTEGER</td>
<td>Override value for MAXPAGES argument on FETCH operator.</td>
</tr>
<tr>
<td>GREEDY</td>
<td>TRUE</td>
<td>Indicates optimizer used greedy algorithm to plan access.</td>
</tr>
<tr>
<td>GLOBLOCK</td>
<td>EXCLUSIVE, INTENT EXCLUSIVE, INTENT NONE, INTENT SHARE, NO LOCK OBTAINED, SHARE, SHARE INTENT EXCLUSIVE, SUPER EXCLUSIVE, UPDATE</td>
<td>Represents global lock intent information for a partitioned table object.</td>
</tr>
<tr>
<td>GROUPBYC</td>
<td>TRUE, FALSE</td>
<td>Whether Group By columns were provided.</td>
</tr>
<tr>
<td>GROUPBYN</td>
<td>Integer</td>
<td>Number of comparison columns.</td>
</tr>
<tr>
<td>GROUPBYR</td>
<td>Each row of this type will contain:</td>
<td>Group By requirement.</td>
</tr>
<tr>
<td></td>
<td>• Ordinal value of column in group by clause (followed by a colon and a space)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Name of column</td>
<td></td>
</tr>
<tr>
<td>HASHCODE</td>
<td>24, 32</td>
<td>Size (in bits) of hash code used for hash join.</td>
</tr>
</tbody>
</table>
### Table 246. ARGUMENT_TYPE and ARGUMENT_VALUE column values (continued)

<table>
<thead>
<tr>
<th>ARGUMENT_TYPE</th>
<th>Possible ARGUMENT_VALUE Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INNERCOL</td>
<td>Each row of this type will contain:</td>
<td>Inner order columns.</td>
</tr>
<tr>
<td></td>
<td>• Ordinal value of column in order (followed by a colon and a space)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Name of column</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Order value</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(A) Ascending</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(D) Descending</td>
<td></td>
</tr>
<tr>
<td>INPUTXID</td>
<td>A context node identifier</td>
<td>INPUTXID identifies the input context node used by the XSCAN operator.</td>
</tr>
<tr>
<td>ISCANMAX</td>
<td>IGNORE INTEGER</td>
<td>Override value for MAXPAGES argument on ISCAN operator.</td>
</tr>
<tr>
<td>JN INPUT</td>
<td>INNER OUTER</td>
<td>Indicates if operator is the operator feeding the inner or outer of a join.</td>
</tr>
<tr>
<td>LCKAVOID</td>
<td>TRUE</td>
<td>Lock avoidance: row access will avoid locking committed data.</td>
</tr>
<tr>
<td>LISTENER</td>
<td>TRUE FALSE</td>
<td>Listener Table Queue indicator.</td>
</tr>
<tr>
<td>MAXPAGES</td>
<td>ALL NONE INTEGER</td>
<td>Maximum pages expected for Prefetch.</td>
</tr>
<tr>
<td>MAXRIDS</td>
<td>NONE INTEGER</td>
<td>Maximum Row Identifiers to be included in each list prefetch request.</td>
</tr>
<tr>
<td>NUMROWS</td>
<td>INTEGER</td>
<td>Number of rows expected to be sorted.</td>
</tr>
<tr>
<td>ONEFETCH</td>
<td>TRUE FALSE</td>
<td>One Fetch indicator.</td>
</tr>
<tr>
<td>OUTERCOL</td>
<td>Each row of this type will contain:</td>
<td>Outer order columns.</td>
</tr>
<tr>
<td></td>
<td>• Ordinal value of column in order (followed by a colon and a space)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Name of column</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Order value</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(A) Ascending</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(D) Descending</td>
<td></td>
</tr>
<tr>
<td>OUTERJN</td>
<td>LEFT RIGHT FULL LEFT (ANTI) RIGHT (ANTI)</td>
<td>Outer join indicator.</td>
</tr>
<tr>
<td>PARTCOLS</td>
<td>Name of Column</td>
<td>Partitioning columns for operator.</td>
</tr>
<tr>
<td>PREFETCH</td>
<td>LIST NONE SEQUENTIAL</td>
<td>Type of Prefetch Eligible.</td>
</tr>
<tr>
<td>REOPT</td>
<td>ALWAYS ONCE</td>
<td>The statement is optimized using bind-in values for parameter markers, host variables, and special registers.</td>
</tr>
</tbody>
</table>
### EXPLAIN_ARGUMENT table

<table>
<thead>
<tr>
<th>ARGUMENT_TYPE</th>
<th>Possible ARGUMENT_VALUE Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RMTQTEXT</td>
<td>Query text</td>
<td>Remote Query Text</td>
</tr>
<tr>
<td>RNG_PROD</td>
<td>Function name</td>
<td>Range producing function for extended index access.</td>
</tr>
<tr>
<td>ROWLOCK</td>
<td>EXCLUSIVE, NONE, REUSE, SHARE, SHORT (INSTANT) SHARE, UPDATE</td>
<td>Row Lock Intent.</td>
</tr>
<tr>
<td>ROWWIDTH</td>
<td>INTEGER</td>
<td>Width of row to be sorted.</td>
</tr>
<tr>
<td>RSUFFIX</td>
<td>Query text</td>
<td>Remote SQL suffix.</td>
</tr>
<tr>
<td>SCANDIR</td>
<td>FORWARD, REVERSE</td>
<td>Scan Direction.</td>
</tr>
<tr>
<td>SCANGRAN</td>
<td>INTEGER</td>
<td>Intra-partition parallelism, granularity of the intra-partition parallel scan, expressed in SCANUNITs.</td>
</tr>
<tr>
<td>SCANSPEED</td>
<td>SLOW, FAST</td>
<td>'SLOW' indicates that the scan is expected to progress slowly over the table. For example, if the scan is the outer of a nested loop join. 'FAST' indicates that the scan is expected to progress with higher speed. This information is used to group scans together for efficient sharing of bufferpool records.</td>
</tr>
<tr>
<td>SCANTYPE</td>
<td>LOCAL, PARALLEL</td>
<td>intra-partition parallelism, Index or Table scan.</td>
</tr>
<tr>
<td>SCANUNIT</td>
<td>ROW, PAGE</td>
<td>Intra-partition parallelism, scan granularity unit.</td>
</tr>
<tr>
<td>SHARED</td>
<td>TRUE</td>
<td>Intra-partition parallelism, shared TEMP indicator.</td>
</tr>
<tr>
<td>SKIP_INS</td>
<td>TRUE</td>
<td>Skip inserted. Row access will skip uncommitted inserted rows. This behavior is enabled with the DB2_SKIPINSERTED registry variable or when currently committed semantics are in effect.</td>
</tr>
<tr>
<td>SKIPDKEY</td>
<td>TRUE</td>
<td>Skip deleted keys. Row access will skip uncommitted deleted keys. This behavior is enabled with the DB2_SKIPDELETED registry variable.</td>
</tr>
<tr>
<td>SKIPDROW</td>
<td>TRUE</td>
<td>Skip deleted rows. Row access will skip uncommitted deleted rows. This behavior is enabled with the DB2_SKIPDELETED registry variable.</td>
</tr>
<tr>
<td>SLOWMAT</td>
<td>TRUE, FALSE</td>
<td>Slow Materialization flag.</td>
</tr>
<tr>
<td>SNGLPROD</td>
<td>TRUE, FALSE</td>
<td>Intra-partition parallelism sort or temp produced by a single agent.</td>
</tr>
</tbody>
</table>
### EXPLAIN_ARGUMENT table

Table 246. ARGUMENT_TYPE and ARGUMENT_VALUE column values (continued)

<table>
<thead>
<tr>
<th>ARGUMENT_TYPE</th>
<th>Possible ARGUMENT_VALUE Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SORTKEY</td>
<td>Each row of this type will contain:</td>
<td>Sort key columns.</td>
</tr>
<tr>
<td></td>
<td>- Ordinal value of column in key (followed by a colon and a space)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Name of column</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Order value</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(A) Ascending</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(D) Descending</td>
<td></td>
</tr>
<tr>
<td>SORTTYPE</td>
<td>PARTITIONED</td>
<td>Intra-partition parallelism, sort type.</td>
</tr>
<tr>
<td></td>
<td>SHARED</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ROUND ROBIN</td>
<td></td>
</tr>
<tr>
<td></td>
<td>REPLICAED</td>
<td></td>
</tr>
<tr>
<td>SRCSEVER</td>
<td>Server name</td>
<td>Source (ship to) server.</td>
</tr>
<tr>
<td>SPILLED</td>
<td>INTEGER</td>
<td>Estimated number of pages in SORT spill.</td>
</tr>
<tr>
<td>SQLCA</td>
<td>Warning information</td>
<td>Warnings and reason codes issued during Explain operation.</td>
</tr>
<tr>
<td>STARJOIN</td>
<td>YES</td>
<td>The IXAND operator is part of a star join</td>
</tr>
<tr>
<td>STMTHEAP</td>
<td>INTEGER</td>
<td>Size of statement heap at start of statement compile.</td>
</tr>
<tr>
<td>STREAM</td>
<td>TRUE</td>
<td>Remote source is streaming.</td>
</tr>
<tr>
<td></td>
<td>FALSE</td>
<td></td>
</tr>
<tr>
<td>TABLOCK</td>
<td>EXCLUSIVE</td>
<td>Table Lock Intent.</td>
</tr>
<tr>
<td></td>
<td>INTENT EXCLUSIVE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>INTENT NONE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>INTENT SHARE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>REUSE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SHARE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SHARE INTENT EXCLUSIVE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SUPER EXCLUSIVE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>UPDATE</td>
<td></td>
</tr>
<tr>
<td>TEMPSIZE</td>
<td>INTEGER</td>
<td>Temporary table page size.</td>
</tr>
<tr>
<td>THROTTLE</td>
<td>TRUE</td>
<td>Throttling improves the performance of other scans that would otherwise lag behind and be forced to reread the same pages. 'TRUE' if the scan can be throttled. 'FALSE' if the scan must not be throttled.</td>
</tr>
<tr>
<td></td>
<td>FALSE</td>
<td></td>
</tr>
<tr>
<td>TMPCMPRS</td>
<td>YES</td>
<td>The value YES indicates that compression is applied. The value ELIGIBLE indicates that compression may be applied if the table becomes large enough. The absence of TMPCMPRS indicates that the temporary table is not compressed.</td>
</tr>
<tr>
<td></td>
<td>ELIGIBLE</td>
<td></td>
</tr>
<tr>
<td>TQDEGREE</td>
<td>INTEGER</td>
<td>Intra-partition parallelism, number of subagents accessing Table Queue.</td>
</tr>
<tr>
<td>TQMERGE</td>
<td>TRUE</td>
<td>Merging (sorted) Table Queue indicator.</td>
</tr>
<tr>
<td></td>
<td>FALSE</td>
<td></td>
</tr>
</tbody>
</table>
## EXPLAIN_ARGUMENT table

<table>
<thead>
<tr>
<th>ARGUMENT_TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
</tr>
<tr>
<td>TQREAD</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>TQSEND</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>TQ TYPE</td>
</tr>
<tr>
<td>TQ_ORIGIN</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>TRUNCTQ</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>TRUNCSRT</td>
</tr>
<tr>
<td>UNIQUE</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>UNIQKEY</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>UR_EXTRA</td>
</tr>
<tr>
<td>VISIBLE</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>VOLATILE</td>
</tr>
<tr>
<td>WRAPPING</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
### EXPLAIN_ARGUMENT table

**Table 246. ARGUMENT_TYPE and ARGUMENT_VALUE column values (continued)**

<table>
<thead>
<tr>
<th>ARGUMENT_TYPE</th>
<th>Possible ARGUMENT_VALUE Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XDFOUT</td>
<td>DECIMAL</td>
<td>XDFOUT indicates the expected number of documents to be returned by the XISCAN operator for each context node.</td>
</tr>
<tr>
<td>XLOGID</td>
<td>An identifier consisting of an SQL schema name and the name of an index over XML data</td>
<td>XLOGID identifies the index over XML data chosen by the optimizer for the XISCAN operator.</td>
</tr>
<tr>
<td>XPATH</td>
<td>An XPATH expression and result set in an internal format</td>
<td>This argument indicates the evaluation of an XPATH expression by the XSCAN operator.</td>
</tr>
<tr>
<td>XPHYID</td>
<td>An identifier consisting of an SQL schema name and the name of a physical index over XML data</td>
<td>XPHYID identifies the physical index that is associated with an index over XML data used by the XISCAN operator.</td>
</tr>
</tbody>
</table>
The EXPLAIN_DIAGNOSTIC table contains an entry for each diagnostic message produced for a particular instance of an explained statement in the EXPLAIN_STATEMENT table. The EXPLAIN_GET_MSGS table function queries the EXPLAIN_DIAGNOSTIC and EXPLAIN_DIAGNOSTIC_DATA Explain tables and returns formatted messages.

Table 247. EXPLAIN_DIAGNOSTIC Table. PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable?</th>
<th>Key?</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXPLAIN_REQUESTER</td>
<td>VARCHAR(128)</td>
<td>No</td>
<td>PK, FK</td>
<td>Authorization ID of initiator of this Explain request.</td>
</tr>
<tr>
<td>EXPLAIN_TIME</td>
<td>TIMESTAMP</td>
<td>No</td>
<td>PK, FK</td>
<td>Time of initiation for Explain request.</td>
</tr>
<tr>
<td>SOURCE_NAME</td>
<td>VARCHAR(128)</td>
<td>No</td>
<td>PK, FK</td>
<td>Name of the package running when the dynamic statement was explained or name of the source file when the static SQL was explained.</td>
</tr>
<tr>
<td>SOURCE_SCHEMA</td>
<td>VARCHAR(128)</td>
<td>No</td>
<td>PK, FK</td>
<td>Schema, or qualifier, of source of Explain request.</td>
</tr>
<tr>
<td>SOURCE_VERSION</td>
<td>VARCHAR(64)</td>
<td>No</td>
<td>PK, FK</td>
<td>Version of the source of the Explain request.</td>
</tr>
<tr>
<td>EXPLAIN_LEVEL</td>
<td>CHAR(1)</td>
<td>No</td>
<td>PK, FK</td>
<td>Level of Explain information for which this row is relevant. Valid values are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>O Original text (as entered by user)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>P PLAN SELECTION</td>
</tr>
<tr>
<td>STMTNO</td>
<td>INTEGER</td>
<td>No</td>
<td>PK, FK</td>
<td>Statement number within package to which this Explain information is related. Set to 1 for dynamic Explain SQL statements. For static SQL statements, this value is the same as the value used for the SYSCAT.STATEMENTS system catalog view.</td>
</tr>
<tr>
<td>SECTNO</td>
<td>INTEGER</td>
<td>No</td>
<td>PK, FK</td>
<td>Section number within package that contains this SQL statement. For dynamic Explain SQL statements, this is the section number used to hold the section for this statement at run time. For static SQL statements, this value is the same as the value used for the SYSCAT.STATEMENTS system catalog view.</td>
</tr>
<tr>
<td>DIAGNOSTIC_ID</td>
<td>INTEGER</td>
<td>No</td>
<td>PK</td>
<td>ID of the diagnostic for a particular instance of a statement in the EXPLAIN_STATEMENT table.</td>
</tr>
<tr>
<td>CODE</td>
<td>INTEGER</td>
<td>No</td>
<td>No</td>
<td>A unique number assigned to each diagnostic message. The number can be used by a message API to retrieve the full text of the diagnostic message.</td>
</tr>
</tbody>
</table>
The EXPLAIN_DIAGNOSTIC_DATA table contains message tokens for specific diagnostic messages that are recorded in the EXPLAIN_DIAGNOSTIC table. The message tokens provide additional information that is specific to the execution of the SQL statement that generated the message.

The EXPLAIN_GET_MSGS table function queries the EXPLAIN_DIAGNOSTIC and EXPLAIN_DIAGNOSTIC_DATA Explain tables, and returns formatted messages.

Table 248. EXPLAIN_DIAGNOSTIC_DATA Table. PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable?</th>
<th>Key?</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXPLAIN_REQUESTER</td>
<td>VARCHAR(128)</td>
<td>No</td>
<td>FK</td>
<td>Authorization ID of initiator of this Explain request.</td>
</tr>
<tr>
<td>EXPLAIN_TIME</td>
<td>TIMESTAMP</td>
<td>No</td>
<td>FK</td>
<td>Time of initiation for Explain request.</td>
</tr>
<tr>
<td>SOURCE_NAME</td>
<td>VARCHAR(128)</td>
<td>No</td>
<td>FK</td>
<td>Name of the package running when the dynamic statement was explained or name of the source file when the static SQL was explained.</td>
</tr>
<tr>
<td>SOURCE_SCHEMA</td>
<td>VARCHAR(128)</td>
<td>No</td>
<td>FK</td>
<td>Schema, or qualifier, of source of Explain request.</td>
</tr>
<tr>
<td>SOURCE_VERSION</td>
<td>VARCHAR(64)</td>
<td>No</td>
<td>FK</td>
<td>Version of the source of the Explain request.</td>
</tr>
<tr>
<td>EXPLAIN_LEVEL</td>
<td>CHAR(1)</td>
<td>No</td>
<td>FK</td>
<td>Level of Explain information for which this row is relevant. Valid values are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>O Original text (as entered by user)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>P PLAN SELECTION</td>
</tr>
<tr>
<td>STMTNO</td>
<td>INTEGER</td>
<td>No</td>
<td>FK</td>
<td>Statement number within package to which this Explain information is related. Set to 1 for dynamic Explain SQL statements. For static SQL statements, this value is the same as the value used for the SYSCAT.STATEMENTS system catalog view.</td>
</tr>
<tr>
<td>SECTNO</td>
<td>INTEGER</td>
<td>No</td>
<td>FK</td>
<td>Section number within package that contains this SQL statement. For dynamic Explain SQL statements, this is the section number used to hold the section for this statement at run time. For static SQL statements, this value is the same as the value used for the SYSCAT.STATEMENTS system catalog view.</td>
</tr>
<tr>
<td>DIAGNOSTIC_ID</td>
<td>INTEGER</td>
<td>No</td>
<td>PK</td>
<td>ID of the diagnostic for a particular instance of a statement in the EXPLAIN_STATEMENT table.</td>
</tr>
<tr>
<td>ORDINAL</td>
<td>INTEGER</td>
<td>No</td>
<td>No</td>
<td>Position of token in the full message text.</td>
</tr>
<tr>
<td>TOKEN</td>
<td>VARCHAR(1000)</td>
<td>Yes</td>
<td>No</td>
<td>Message token to be inserted into the full message text; might be truncated.</td>
</tr>
<tr>
<td>TOKEN_LONG</td>
<td>BLOB(3M)</td>
<td>Yes</td>
<td>No</td>
<td>More detailed information, if available.</td>
</tr>
</tbody>
</table>
**EXPLAIN_INSTANCE table**

The EXPLAIN_INSTANCE table is the main control table for all Explain information. Each row of data in the Explain tables is explicitly linked to one unique row in this table. The EXPLAIN_INSTANCE table gives basic information about the source of the SQL statements being explained as well as information about the environment in which the explanation took place.

*Table 249. EXPLAIN_INSTANCE Table.* PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable?</th>
<th>Key?</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXPLAIN_REQUESTER</td>
<td>VARCHAR(128)</td>
<td>No</td>
<td>PK</td>
<td>Authorization ID of initiator of this Explain request.</td>
</tr>
<tr>
<td>EXPLAIN_TIME</td>
<td>TIMESTAMP</td>
<td>No</td>
<td>PK</td>
<td>Time of initiation for Explain request.</td>
</tr>
<tr>
<td>SOURCE_NAME</td>
<td>VARCHAR(128)</td>
<td>No</td>
<td>PK</td>
<td>Name of the package running when the dynamic statement was explained or name of the source file when the static SQL was explained.</td>
</tr>
<tr>
<td>SOURCE_SCHEMA</td>
<td>VARCHAR(128)</td>
<td>No</td>
<td>PK</td>
<td>Schema, or qualifier, of source of Explain request.</td>
</tr>
<tr>
<td>SOURCE_VERSION</td>
<td>VARCHAR(64)</td>
<td>No</td>
<td>PK</td>
<td>Version of the source of the Explain request.</td>
</tr>
<tr>
<td>EXPLAIN_OPTION</td>
<td>CHAR(1)</td>
<td>No</td>
<td>No</td>
<td>Indicates what Explain Information was requested for this request.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Possible values are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>P PLAN SELECTION</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>S Section Explain</td>
</tr>
<tr>
<td>SNAPSHOT_TAKEN</td>
<td>CHAR(1)</td>
<td>No</td>
<td>No</td>
<td>Indicates whether an Explain Snapshot was taken for this request.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Possible values are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Y Yes, an Explain Snapshot(s) was taken and stored in the EXPLAIN_STATEMENT table. Regular Explain information was also captured.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>N No Explain Snapshot was taken. Regular Explain information was captured.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>O Only an Explain Snapshot was taken. Regular Explain information was not captured.</td>
</tr>
<tr>
<td>DB2_VERSION</td>
<td>CHAR(7)</td>
<td>No</td>
<td>No</td>
<td>Release number for the DB2 product that processed this explain request. Format is <em>vv.rr.m</em>, where:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vv Version number</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>rr Release number</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>m Maintenance release number</td>
</tr>
<tr>
<td>SQL_TYPE</td>
<td>CHAR(1)</td>
<td>No</td>
<td>No</td>
<td>Indicates whether the Explain Instance was for static or dynamic SQL.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Possible values are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>S Static SQL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>D Dynamic SQL</td>
</tr>
</tbody>
</table>
### EXPLAIN_INSTANCE table

**Table 249. EXPLAIN_INSTANCE Table (continued).** PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable?</th>
<th>Key?</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>QUERYOPT</td>
<td>INTEGER</td>
<td>No</td>
<td>No</td>
<td>Indicates the query optimization class used by the SQL Compiler at the time of the Explain invocation. The value indicates what level of query optimization was performed by the SQL Compiler for the SQL statements being explained.</td>
</tr>
<tr>
<td>BLOCK</td>
<td>CHAR(1)</td>
<td>No</td>
<td>No</td>
<td>Indicates what type of cursor blocking was used when compiling the SQL statements. For more information, see the BLOCK column in SYSCAT.PACKAGES. Possible values are: N No Blocking U Block Unambiguous Cursors B Block All Cursors</td>
</tr>
<tr>
<td>ISOLATION</td>
<td>CHAR(2)</td>
<td>No</td>
<td>No</td>
<td>Indicates what type of isolation was used when compiling the SQL statements. For more information, see the ISOLATION column in SYSCAT.PACKAGES. Possible values are: RR Repeatable Read RS Read Stability CS Cursor Stability UR Uncommitted Read</td>
</tr>
<tr>
<td>BUFFPAGE</td>
<td>INTEGER</td>
<td>No</td>
<td>No</td>
<td>Contains the value of the BUFFPAGE database configuration setting at the time of the Explain invocation.</td>
</tr>
<tr>
<td>AVG_APPLS</td>
<td>INTEGER</td>
<td>No</td>
<td>No</td>
<td>Contains the value of the avg_appls configuration parameter at the time of the Explain invocation.</td>
</tr>
<tr>
<td>SORTHEAP</td>
<td>INTEGER</td>
<td>No</td>
<td>No</td>
<td>Contains the value of the sortheap database configuration parameter at the time of the Explain invocation.</td>
</tr>
<tr>
<td>LOCKLIST</td>
<td>INTEGER</td>
<td>No</td>
<td>No</td>
<td>Contains the value of the locklist database configuration parameter at the time of the Explain invocation.</td>
</tr>
<tr>
<td>MAXLOCKS</td>
<td>SMALLINT</td>
<td>No</td>
<td>No</td>
<td>Contains the value of the maxlocks database configuration parameter at the time of the Explain invocation.</td>
</tr>
<tr>
<td>LOCKS_AVAIL</td>
<td>INTEGER</td>
<td>No</td>
<td>No</td>
<td>Contains the number of locks assumed to be available by the optimizer for each user. (Derived from locklist and maxlocks.)</td>
</tr>
<tr>
<td>CPU_SPEED</td>
<td>DOUBLE</td>
<td>No</td>
<td>No</td>
<td>Contains the value of the cpuspeed database manager configuration parameter at the time of the Explain invocation.</td>
</tr>
<tr>
<td>REMARKS</td>
<td>VARCHAR(254)</td>
<td>Yes</td>
<td>No</td>
<td>User-provided comment.</td>
</tr>
<tr>
<td>DBHEAP</td>
<td>INTEGER</td>
<td>No</td>
<td>No</td>
<td>Contains the value of the dbheap database configuration parameter at the time of Explain invocation.</td>
</tr>
</tbody>
</table>
**EXPLAIN_INSTANCE table**

*Table 249. EXPLAIN_INSTANCE Table (continued).* PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable?</th>
<th>Key?</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMM_SPEED</td>
<td>DOUBLE</td>
<td>No</td>
<td>No</td>
<td>Contains the value of the <code>comm_bandwidth</code> database configuration parameter at the time of Explain invocation.</td>
</tr>
<tr>
<td>PARALLELISM</td>
<td>CHAR(2)</td>
<td>No</td>
<td>No</td>
<td>Possible values are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• N = No parallelism</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• P = Intra-partition parallelism</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• IP = Inter-partition parallelism</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• BP = Intra-partition parallelism and inter-partition parallelism</td>
</tr>
<tr>
<td>DATAJOINER</td>
<td>CHAR(1)</td>
<td>No</td>
<td>No</td>
<td>Possible values are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• N = Non-federated systems plan</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Y = Federated systems plan</td>
</tr>
</tbody>
</table>
**EXPLAIN_OBJECT table**

The EXPLAIN_OBJECT table identifies those data objects required by the access plan generated to satisfy the SQL statement.

**Table 250. EXPLAIN_OBJECT Table.** PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable?</th>
<th>Key?</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXPLAIN_REQUESTER</td>
<td>VARCHAR(128)</td>
<td>No</td>
<td>FK</td>
<td>Authorization ID of initiator of this Explain request.</td>
</tr>
<tr>
<td>EXPLAIN_TIME</td>
<td>TIMESTAMP</td>
<td>No</td>
<td>FK</td>
<td>Time of initiation for Explain request.</td>
</tr>
<tr>
<td>SOURCE_NAME</td>
<td>VARCHAR(128)</td>
<td>No</td>
<td>FK</td>
<td>Name of the package running when the dynamic statement was explained or name of the source file when the static SQL was explained.</td>
</tr>
<tr>
<td>SOURCE_SCHEMA</td>
<td>VARCHAR(128)</td>
<td>No</td>
<td>FK</td>
<td>Schema, or qualifier, of source of Explain request.</td>
</tr>
<tr>
<td>SOURCE_VERSION</td>
<td>VARCHAR(64)</td>
<td>No</td>
<td>FK</td>
<td>Version of the source of the Explain request.</td>
</tr>
<tr>
<td>EXPLAIN_LEVEL</td>
<td>CHAR(1)</td>
<td>No</td>
<td>FK</td>
<td>Level of Explain information for which this row is relevant.</td>
</tr>
<tr>
<td>STMTNO</td>
<td>INTEGER</td>
<td>No</td>
<td>FK</td>
<td>Statement number within package to which this explain information is related.</td>
</tr>
<tr>
<td>SECTNO</td>
<td>INTEGER</td>
<td>No</td>
<td>FK</td>
<td>Section number within package to which this explain information is related.</td>
</tr>
<tr>
<td>OBJECT_SCHEMA</td>
<td>VARCHAR(128)</td>
<td>No</td>
<td>No</td>
<td>Schema to which this object belongs.</td>
</tr>
<tr>
<td>OBJECT_NAME</td>
<td>VARCHAR(128)</td>
<td>No</td>
<td>No</td>
<td>Name of the object.</td>
</tr>
<tr>
<td>OBJECT_TYPE</td>
<td>CHAR(2)</td>
<td>No</td>
<td>No</td>
<td>Descriptive label for the type of object.</td>
</tr>
<tr>
<td>CREATE_TIME</td>
<td>TIMESTAMP</td>
<td>Yes</td>
<td>No</td>
<td>Time of Object’s creation; null if a table function.</td>
</tr>
<tr>
<td>STATISTICS_TIME</td>
<td>TIMESTAMP</td>
<td>Yes</td>
<td>No</td>
<td>Last time of update to statistics for this object; null if statistics do not exist for this object.</td>
</tr>
<tr>
<td>COLUMN_COUNT</td>
<td>SMALLINT</td>
<td>No</td>
<td>No</td>
<td>Number of columns in this object.</td>
</tr>
<tr>
<td>ROW_COUNT</td>
<td>INTEGER</td>
<td>No</td>
<td>No</td>
<td>Estimated number of rows in this object.</td>
</tr>
<tr>
<td>WIDTH</td>
<td>INTEGER</td>
<td>No</td>
<td>No</td>
<td>The average width of the object in bytes. Set to -1 for an index.</td>
</tr>
<tr>
<td>PAGES</td>
<td>BIGINT</td>
<td>No</td>
<td>No</td>
<td>Estimated number of pages that the object occupies in the buffer pool. Set to -1 for a table function.</td>
</tr>
<tr>
<td>DISTINCT</td>
<td>CHAR(1)</td>
<td>No</td>
<td>No</td>
<td>Indicates whether the rows in the object are distinct (that is, whether there are duplicates).</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Possible values are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Y    Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>N    No</td>
</tr>
<tr>
<td>TABLESPACE_NAME</td>
<td>VARCHAR(128)</td>
<td>Yes</td>
<td>No</td>
<td>Name of the table space in which this object is stored; set to null if no table space is involved.</td>
</tr>
<tr>
<td>OVERHEAD</td>
<td>DOUBLE</td>
<td>No</td>
<td>No</td>
<td>Total estimated overhead, in milliseconds, for a single random I/O to the specified table space. Includes controller overhead, disk seek, and latency times. Set to -1 if no table space is involved.</td>
</tr>
</tbody>
</table>
### EXPLAIN_OBJECT Table (continued)

**PK** means that the column is part of a primary key; **FK** means that the column is part of a foreign key.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable?</th>
<th>Key?</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRANSFER_RATE</td>
<td>DOUBLE</td>
<td>No</td>
<td>No</td>
<td>Estimated time to read a data page, in milliseconds, from the specified table space. Set to -1 if no table space is involved.</td>
</tr>
<tr>
<td>PREFETCHSIZE</td>
<td>INTEGER</td>
<td>No</td>
<td>No</td>
<td>Number of data pages to be read when prefetch is performed. Set to -1 for a table function.</td>
</tr>
<tr>
<td>EXTENTSIZE</td>
<td>INTEGER</td>
<td>No</td>
<td>No</td>
<td>Size of extent, in data pages. This many pages are written to one container in the table space before switching to the next container. Set to -1 for a table function.</td>
</tr>
<tr>
<td>CLUSTER</td>
<td>DOUBLE</td>
<td>No</td>
<td>No</td>
<td>Degree of data clustering with the index. If &gt;= 1, this is the CLUSTERRATIO. If &gt;= 0 and &lt; 1, this is the CLUSTERFACTOR. Set to -1 for a table, table function, or if this statistic is not available.</td>
</tr>
<tr>
<td>NLEAF</td>
<td>BIGINT</td>
<td>No</td>
<td>No</td>
<td>Number of leaf pages this index object’s values occupy. Set to -1 for a table, table function, or if this statistic is not available.</td>
</tr>
<tr>
<td>NLEVELS</td>
<td>INTEGER</td>
<td>No</td>
<td>No</td>
<td>Number of index levels in this index object’s tree. Set to -1 for a table, table function, or if this statistic is not available.</td>
</tr>
<tr>
<td>FULLKEYCARD</td>
<td>BIGINT</td>
<td>No</td>
<td>No</td>
<td>Number of distinct full key values contained in this index object. Set to -1 for a table, table function, or if this statistic is not available.</td>
</tr>
<tr>
<td>OVERFLOW</td>
<td>BIGINT</td>
<td>No</td>
<td>No</td>
<td>Total number of overflow records in the table. Set to -1 for an index, table function, or if this statistic is not available.</td>
</tr>
<tr>
<td>FIRSTKEYCARD</td>
<td>BIGINT</td>
<td>No</td>
<td>No</td>
<td>Number of distinct first key values. Set to -1 for a table, table function, or if this statistic is not available.</td>
</tr>
<tr>
<td>FIRST2KEYCARD</td>
<td>BIGINT</td>
<td>No</td>
<td>No</td>
<td>Number of distinct first key values using the first {2,3,4} columns of the index. Set to -1 for a table, table function, or if this statistic is not available.</td>
</tr>
<tr>
<td>FIRST3KEYCARD</td>
<td>BIGINT</td>
<td>No</td>
<td>No</td>
<td>Number of distinct first key values using the first {2,3,4} columns of the index. Set to -1 for a table, table function, or if this statistic is not available.</td>
</tr>
<tr>
<td>FIRST4KEYCARD</td>
<td>BIGINT</td>
<td>No</td>
<td>No</td>
<td>Number of leaf pages located on disk in index key order with few or no large gaps between them. Set to -1 for a table, table function, or if this statistic is not available.</td>
</tr>
<tr>
<td>SEQUENTIAL_PAGES</td>
<td>BIGINT</td>
<td>No</td>
<td>No</td>
<td>Average number of index pages accessible in sequence.</td>
</tr>
<tr>
<td>DENSITY</td>
<td>INTEGER</td>
<td>No</td>
<td>No</td>
<td>Ratio of SEQUENTIAL_PAGES to number of pages in the range of pages occupied by the index, expressed as a percentage (integer between 0 and 100). Set to -1 for a table, table function, or if this statistic is not available.</td>
</tr>
<tr>
<td>STATS_SRC</td>
<td>CHAR(1)</td>
<td>No</td>
<td>No</td>
<td>Indicates the source for the statistics. Set to 1 if from single node.</td>
</tr>
<tr>
<td>AVERAGE_SEQUENCE_</td>
<td>DOUBLE</td>
<td>No</td>
<td>No</td>
<td>Gap between sequences.</td>
</tr>
<tr>
<td>GAP</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AVERAGE_SEQUENCE_</td>
<td>DOUBLE</td>
<td>No</td>
<td>No</td>
<td>Gap between sequences when fetching using the index.</td>
</tr>
<tr>
<td>FETCH_GAP</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AVERAGE_SEQUENCE_</td>
<td>DOUBLE</td>
<td>No</td>
<td>No</td>
<td>Average number of index pages accessible in sequence.</td>
</tr>
</tbody>
</table>
### EXPLAIN_OBJECT table

*Table 250. EXPLAIN_OBJECT Table (continued)*. PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable?</th>
<th>Key?</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVERAGE_SEQUENCE_FETCH_PAGES</td>
<td>DOUBLE</td>
<td>No</td>
<td>No</td>
<td>Average number of table pages accessible in sequence when fetching using the index.</td>
</tr>
<tr>
<td>AVERAGE_RANDOM_PAGES</td>
<td>DOUBLE</td>
<td>No</td>
<td>No</td>
<td>Average number of random index pages between sequential page accesses.</td>
</tr>
<tr>
<td>AVERAGE_RANDOM_FETCH_PAGES</td>
<td>DOUBLE</td>
<td>No</td>
<td>No</td>
<td>Average number of random table pages between sequential page accesses when fetching using the index.</td>
</tr>
<tr>
<td>NUMRIDS</td>
<td>BIGINT</td>
<td>No</td>
<td>No</td>
<td>Total number of row identifiers in the index.</td>
</tr>
<tr>
<td>NUMRIDS_DELETED</td>
<td>BIGINT</td>
<td>No</td>
<td>No</td>
<td>Total number of pseudo-deleted row identifiers in the index.</td>
</tr>
<tr>
<td>NUM_EMPTY_LEAFS</td>
<td>BIGINT</td>
<td>No</td>
<td>No</td>
<td>Total number of empty leaf pages in the index.</td>
</tr>
<tr>
<td>ACTIVE_BLOCKS</td>
<td>BIGINT</td>
<td>No</td>
<td>No</td>
<td>Total number of active multidimensional clustering (MDC) blocks in the table.</td>
</tr>
<tr>
<td>NUM_DATA_PART</td>
<td>INTEGER</td>
<td>No</td>
<td>No</td>
<td>Number of data partitions for a partitioned table. Set to 1 if the table is not partitioned.</td>
</tr>
</tbody>
</table>

*Table 251. Possible OBJECT_TYPE Values*

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IX</td>
<td>Index</td>
</tr>
<tr>
<td>NK</td>
<td>Nickname</td>
</tr>
<tr>
<td>RX</td>
<td>RCT Index</td>
</tr>
<tr>
<td>DP</td>
<td>Data partitioned table</td>
</tr>
<tr>
<td>TA</td>
<td>Table</td>
</tr>
<tr>
<td>TF</td>
<td>Table Function</td>
</tr>
<tr>
<td>+A</td>
<td>Compiler-referenced Alias</td>
</tr>
<tr>
<td>+C</td>
<td>Compiler-referenced Constraint</td>
</tr>
<tr>
<td>+F</td>
<td>Compiler-referenced Function</td>
</tr>
<tr>
<td>+G</td>
<td>Compiler-referenced Trigger</td>
</tr>
<tr>
<td>+N</td>
<td>Compiler-referenced Nickname</td>
</tr>
<tr>
<td>+T</td>
<td>Compiler-referenced Table</td>
</tr>
<tr>
<td>+V</td>
<td>Compiler-referenced View</td>
</tr>
<tr>
<td>XI</td>
<td>Logical XML index</td>
</tr>
<tr>
<td>PI</td>
<td>Physical XML index</td>
</tr>
<tr>
<td>LI</td>
<td>Partitioned index</td>
</tr>
<tr>
<td>LX</td>
<td>Partitioned logical XML index</td>
</tr>
<tr>
<td>LP</td>
<td>Partitioned physical XML index</td>
</tr>
</tbody>
</table>
The EXPLAIN_OPERATOR table contains all the operators needed to satisfy the query statement by the query compiler.

Table 252. EXPLAIN_OPERATOR Table. PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable?</th>
<th>Key?</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXPLAIN_REQUESTER</td>
<td>VARCHAR(128)</td>
<td>No</td>
<td>PK</td>
<td>Authorization ID of initiator of this Explain request.</td>
</tr>
<tr>
<td>EXPLAIN_TIME</td>
<td>TIMESTAMP</td>
<td>No</td>
<td>PK</td>
<td>Time of initiation for Explain request.</td>
</tr>
<tr>
<td>SOURCE_NAME</td>
<td>VARCHAR(128)</td>
<td>No</td>
<td>PK</td>
<td>Name of the package running when the dynamic statement was explained or name of the source file when the static SQL was explained.</td>
</tr>
<tr>
<td>SOURCE_SCHEMA</td>
<td>VARCHAR(128)</td>
<td>No</td>
<td>PK</td>
<td>Schema, or qualifier, of source of Explain request.</td>
</tr>
<tr>
<td>SOURCE_VERSION</td>
<td>VARCHAR(64)</td>
<td>No</td>
<td>PK</td>
<td>Version of the source of the Explain request.</td>
</tr>
<tr>
<td>EXPLAIN_LEVEL</td>
<td>CHAR(1)</td>
<td>No</td>
<td>PK</td>
<td>Level of Explain information for which this row is relevant.</td>
</tr>
<tr>
<td>STMTNO</td>
<td>INTEGER</td>
<td>No</td>
<td>PK</td>
<td>Statement number within package to which this explain information is related.</td>
</tr>
<tr>
<td>SECTNO</td>
<td>INTEGER</td>
<td>No</td>
<td>PK</td>
<td>Section number within package to which this explain information is related.</td>
</tr>
<tr>
<td>OPERATOR_ID</td>
<td>INTEGER</td>
<td>No</td>
<td>PK</td>
<td>Unique ID for this operator within this query.</td>
</tr>
<tr>
<td>OPERATOR_TYPE</td>
<td>CHAR(6)</td>
<td>No</td>
<td>No</td>
<td>Descriptive label for the type of operator.</td>
</tr>
<tr>
<td>TOTAL_COST</td>
<td>DOUBLE</td>
<td>No</td>
<td>No</td>
<td>Estimated cumulative total cost (in timerons) of executing the chosen access plan up to and including this operator.</td>
</tr>
<tr>
<td>IO_COST</td>
<td>DOUBLE</td>
<td>No</td>
<td>No</td>
<td>Estimated cumulative I/O cost (in data page I/Os) of executing the chosen access plan up to and including this operator.</td>
</tr>
<tr>
<td>CPU_COST</td>
<td>DOUBLE</td>
<td>No</td>
<td>No</td>
<td>Estimated cumulative CPU cost (in instructions) of executing the chosen access plan up to and including this operator.</td>
</tr>
<tr>
<td>FIRST_ROW_COST</td>
<td>DOUBLE</td>
<td>No</td>
<td>No</td>
<td>Estimated cumulative cost (in timerons) of fetching the first row for the access plan up to and including this operator. This value includes any initial overhead required.</td>
</tr>
<tr>
<td>RE_TOTAL_COST</td>
<td>DOUBLE</td>
<td>No</td>
<td>No</td>
<td>Estimated cumulative cost (in timerons) of fetching the next row for the chosen access plan up to and including this operator.</td>
</tr>
<tr>
<td>RE_IO_COST</td>
<td>DOUBLE</td>
<td>No</td>
<td>No</td>
<td>Estimated cumulative I/O cost (in data page I/Os) of fetching the next row for the chosen access plan up to and including this operator.</td>
</tr>
<tr>
<td>RE_CPU_COST</td>
<td>DOUBLE</td>
<td>No</td>
<td>No</td>
<td>Estimated cumulative CPU cost (in instructions) of fetching the next row for the chosen access plan up to and including this operator.</td>
</tr>
<tr>
<td>COMM_COST</td>
<td>DOUBLE</td>
<td>No</td>
<td>No</td>
<td>Estimated cumulative communication cost (in TCP/IP frames) of executing the chosen access plan up to and including this operator.</td>
</tr>
</tbody>
</table>
**EXPLAIN_OPERATOR table**

*Table 252. EXPLAIN_OPERATOR Table (continued).* PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable?</th>
<th>Key?</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIRST_COMM_COST</td>
<td>DOUBLE</td>
<td>No</td>
<td>No</td>
<td>Estimated cumulative communications cost (in TCP/IP frames) of fetching the first row for the chosen access plan up to and including this operator. This value includes any initial overhead required.</td>
</tr>
<tr>
<td>BUFFERS</td>
<td>DOUBLE</td>
<td>No</td>
<td>No</td>
<td>Estimated buffer requirements for this operator and its inputs.</td>
</tr>
<tr>
<td>REMOTE_TOTAL_COST</td>
<td>DOUBLE</td>
<td>No</td>
<td>No</td>
<td>Estimated cumulative total cost (in timerons) of performing operation(s) on remote database(s).</td>
</tr>
<tr>
<td>REMOTE_COMM_COST</td>
<td>DOUBLE</td>
<td>No</td>
<td>No</td>
<td>Estimated cumulative communication cost of executing the chosen remote access plan up to and including this operator.</td>
</tr>
</tbody>
</table>

*Table 253. OPERATOR_TYPE values*

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DELETE</td>
<td>Delete</td>
</tr>
<tr>
<td>EISCAN</td>
<td>Extended Index Scan</td>
</tr>
<tr>
<td>FETCH</td>
<td>Fetch</td>
</tr>
<tr>
<td>FILTER</td>
<td>Filter rows</td>
</tr>
<tr>
<td>GENROW</td>
<td>Generate Row</td>
</tr>
<tr>
<td>GRPBY</td>
<td>Group By</td>
</tr>
<tr>
<td>HSJOIN</td>
<td>Hash Join</td>
</tr>
<tr>
<td>INSERT</td>
<td>Insert</td>
</tr>
<tr>
<td>IXAND</td>
<td>Dynamic Bitmap Index ANDing</td>
</tr>
<tr>
<td>IXSCAN</td>
<td>Relational index scan</td>
</tr>
<tr>
<td>MSJOIN</td>
<td>Merge Scan Join</td>
</tr>
<tr>
<td>NLJOIN</td>
<td>Nested loop Join</td>
</tr>
<tr>
<td>RETURN</td>
<td>Result</td>
</tr>
<tr>
<td>RIDSCN</td>
<td>Row Identifier (RID) Scan</td>
</tr>
<tr>
<td>RPD</td>
<td>Remote PushDown</td>
</tr>
<tr>
<td>SHIP</td>
<td>Ship query to remote system</td>
</tr>
<tr>
<td>SORT</td>
<td>Sort</td>
</tr>
<tr>
<td>TBSCAN</td>
<td>Table Scan</td>
</tr>
<tr>
<td>TEMP</td>
<td>Temporary Table Construction</td>
</tr>
<tr>
<td>TQ</td>
<td>Table Queue</td>
</tr>
<tr>
<td>UNION</td>
<td>Union</td>
</tr>
<tr>
<td>UNIQUE</td>
<td>Duplicate Elimination</td>
</tr>
<tr>
<td>UPDATE</td>
<td>Update</td>
</tr>
<tr>
<td>XISCAN</td>
<td>Index scan over XML data</td>
</tr>
<tr>
<td>XSCAN</td>
<td>XML document navigation scan</td>
</tr>
<tr>
<td>XANDOR</td>
<td>Index ANDing and ORing over XML data</td>
</tr>
</tbody>
</table>
The EXPLAIN_PREDICATE table identifies which predicates are applied by a specific operator.

**Table 254. EXPLAIN_PREDICATE Table.** PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable?</th>
<th>Key?</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXPLAIN_REQUESTER</td>
<td>VARCHAR(128)</td>
<td>No</td>
<td>FK</td>
<td>Authorization ID of initiator of this Explain request.</td>
</tr>
<tr>
<td>EXPLAIN_TIME</td>
<td>TIMESTAMP</td>
<td>No</td>
<td>FK</td>
<td>Time of initiation for Explain request.</td>
</tr>
<tr>
<td>SOURCE_NAME</td>
<td>VARCHAR(128)</td>
<td>No</td>
<td>FK</td>
<td>Name of the package running when the dynamic statement was explained or name of the source file when the static SQL was explained.</td>
</tr>
<tr>
<td>SOURCE_SCHEMA</td>
<td>VARCHAR(128)</td>
<td>No</td>
<td>FK</td>
<td>Schema, or qualifier, of source of Explain request.</td>
</tr>
<tr>
<td>SOURCE_VERSION</td>
<td>VARCHAR(64)</td>
<td>No</td>
<td>FK</td>
<td>Version of the source of the Explain request.</td>
</tr>
<tr>
<td>EXPLAIN_LEVEL</td>
<td>CHAR(1)</td>
<td>No</td>
<td>FK</td>
<td>Level of Explain information for which this row is relevant.</td>
</tr>
<tr>
<td>STMTNO</td>
<td>INTEGER</td>
<td>No</td>
<td>FK</td>
<td>Statement number within package to which this explain information is related.</td>
</tr>
<tr>
<td>SECTNO</td>
<td>INTEGER</td>
<td>No</td>
<td>FK</td>
<td>Section number within package to which this explain information is related.</td>
</tr>
<tr>
<td>OPERATOR_ID</td>
<td>INTEGER</td>
<td>No</td>
<td>No</td>
<td>Unique ID for this operator within this query.</td>
</tr>
<tr>
<td>PREDICATE_ID</td>
<td>INTEGER</td>
<td>No</td>
<td>No</td>
<td>Unique ID for this predicate for the specified operator.</td>
</tr>
<tr>
<td>HOW_APPLIED</td>
<td>CHAR(10)</td>
<td>No</td>
<td>No</td>
<td>How predicate is being used by the specified operator.</td>
</tr>
<tr>
<td>WHEN_EVALUATED</td>
<td>CHAR(3)</td>
<td>No</td>
<td>No</td>
<td>Indicates when the subquery used in this predicate is evaluated.</td>
</tr>
<tr>
<td>RELOP_TYPE</td>
<td>CHAR(2)</td>
<td>No</td>
<td>No</td>
<td>The type of relational operator used in this predicate.</td>
</tr>
</tbody>
</table>

Possible values are:
- **blank**  This predicate does not contain a subquery.
- **EAA** The subquery used in this predicate is evaluated at application (EAA). That is, it is re-evaluated for every row processed by the specified operator, as the predicate is being applied.
- **EAO** The subquery used in this predicate is evaluated at open (EAO). That is, it is re-evaluated only once for the specified operator, and its results are re-used in the application of the predicate for each row.
- **MUL** There is more than one type of subquery in this predicate.
### EXPLAIN_PREDICATE table

**Table 254. EXPLAIN_PREDICATE Table (continued).** PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable?</th>
<th>Key?</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUBQUERY</td>
<td>CHAR(1)</td>
<td>No</td>
<td>No</td>
<td>Whether or not a data stream from a subquery is required for this predicate. There may be multiple subquery streams required. Possible values are: N: No subquery stream is required, Y: One or more subquery streams is required.</td>
</tr>
<tr>
<td>FILTER_FACTOR</td>
<td>DOUBLE</td>
<td>No</td>
<td>No</td>
<td>The estimated fraction of rows that will be qualified by this predicate. A value of &quot;-1&quot; is shown when FILTER_FACTOR is not applicable. FILTER_FACTOR is not applicable for operator predicates constructed by the Explain tool which are not optimizer objects and do not exist in the optimizer plan.</td>
</tr>
<tr>
<td>PREDICATE_TEXT</td>
<td>CLOB(2M)</td>
<td>Yes</td>
<td>No</td>
<td>The text of the predicate as recreated from the internal representation of the SQL or XQuery statement. If the value of a host variable, special register, or parameter marker is used during compilation of the statement, this value will appear at the end of the predicate text enclosed in a comment. The value will be stored in the EXPLAIN_PREDICATE table only if the statement is executed by a user who has DBADM authority, or if the DB2 registry variable DB2_VIEW_REOPT_VALUES is set to YES; otherwise, an empty comment will appear at the end of the predicate text. Null if not available.</td>
</tr>
<tr>
<td>RANGE_NUM</td>
<td>INTEGER</td>
<td>Yes</td>
<td>No</td>
<td>Range of data partition elimination predicates, which enables the grouping of predicates that are used for data partition elimination by range. Null value for all other predicate types.</td>
</tr>
</tbody>
</table>

**Table 255. Possible HOW_APPLIED Values**

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BSARG</td>
<td>Evaluated as a sargable predicate once for every block</td>
</tr>
<tr>
<td>DPSTART</td>
<td>Start key predicate used in data partition elimination</td>
</tr>
<tr>
<td>DPSTOP</td>
<td>Stop key predicate used in data partition elimination</td>
</tr>
<tr>
<td>JOIN</td>
<td>Used to join tables</td>
</tr>
<tr>
<td>RESID</td>
<td>Evaluated as a residual predicate</td>
</tr>
<tr>
<td>SARG</td>
<td>Evaluated as a sargable predicate for index or data page</td>
</tr>
<tr>
<td>START</td>
<td>Used as a start condition</td>
</tr>
<tr>
<td>STOP</td>
<td>Used as a stop condition</td>
</tr>
</tbody>
</table>
### EXPLAIN_PREDICATE table

Table 256. Possible RELOP_TYPE Values

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>blanks</td>
<td>Not Applicable</td>
</tr>
<tr>
<td>EQ</td>
<td>Equals</td>
</tr>
<tr>
<td>GE</td>
<td>Greater Than or Equal</td>
</tr>
<tr>
<td>GT</td>
<td>Greater Than</td>
</tr>
<tr>
<td>IN</td>
<td>In list</td>
</tr>
<tr>
<td>LE</td>
<td>Less Than or Equal</td>
</tr>
<tr>
<td>LK</td>
<td>Like</td>
</tr>
<tr>
<td>LT</td>
<td>Less Than</td>
</tr>
<tr>
<td>NE</td>
<td>Not Equal</td>
</tr>
<tr>
<td>NL</td>
<td>Is Null</td>
</tr>
<tr>
<td>NN</td>
<td>Is Not Null</td>
</tr>
</tbody>
</table>
EXPLAIN_STATEMENT table

The EXPLAIN_STATEMENT table contains the text of the SQL statement as it exists for the different levels of Explain information. The original SQL statement as entered by the user is stored in this table along with the version used (by the optimizer) to choose an access plan to satisfy the SQL statement. The latter version may bear little resemblance to the original as it may have been rewritten and/or enhanced with additional predicates as determined by the SQL Compiler. In addition, if statement concentrator is enabled and the statement was changed as a result of statement concentrator, the effective SQL statement will also be stored in this table. This statement will resemble the original statement except that the literal values will be replaced with system generated named parameter markers. The plan information will be based on the effective statement in this case.

Table 257. EXPLAIN_STATEMENT Table. PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable?</th>
<th>Key?</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXPLAIN_REQUESTER</td>
<td>VARCHAR(128)</td>
<td>No</td>
<td>PK, FK</td>
<td>Authorization ID of initiator of this Explain request.</td>
</tr>
<tr>
<td>EXPLAIN_TIME</td>
<td>TIMESTAMP</td>
<td>No</td>
<td>PK, FK</td>
<td>Time of initiation for Explain request.</td>
</tr>
<tr>
<td>SOURCE_NAME</td>
<td>VARCHAR(128)</td>
<td>No</td>
<td>PK, FK</td>
<td>Name of the package running when the dynamic statement was explained or name of the source file when the static SQL was explained.</td>
</tr>
<tr>
<td>SOURCE_SCHEMA</td>
<td>VARCHAR(128)</td>
<td>No</td>
<td>PK, FK</td>
<td>Schema, or qualifier, of source of Explain request.</td>
</tr>
<tr>
<td>SOURCE_VERSION</td>
<td>VARCHAR(64)</td>
<td>No</td>
<td>FK</td>
<td>Version of the source of the Explain request.</td>
</tr>
<tr>
<td>EXPLAIN_LEVEL</td>
<td>CHAR(1)</td>
<td>No</td>
<td>PK</td>
<td>Level of Explain information for which this row is relevant.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Valid values are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>E    Effective SQL text</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>O    Original Text (as entered by user)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>P    PLAN SELECTION</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>S    Section Explain</td>
</tr>
<tr>
<td>STMTNO</td>
<td>INTEGER</td>
<td>No</td>
<td>PK</td>
<td>Statement number within package to which this explain information is related. Set to 1 for dynamic Explain SQL statements. For static SQL statements, this value is the same as the value used for the SYSCAT.STATEMENTS catalog view.</td>
</tr>
<tr>
<td>SECTNO</td>
<td>INTEGER</td>
<td>No</td>
<td>PK</td>
<td>Section number within package that contains this SQL statement. For dynamic Explain SQL statements, this is the section number used to hold the section for this statement at runtime. For static SQL statements, this value is the same as the value used for the SYSCAT.STATEMENTS catalog view.</td>
</tr>
</tbody>
</table>
**EXPLAIN_STATEMENT table**

*Table 257. EXPLAIN_STATEMENT Table (continued).* PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable?</th>
<th>Key?</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>QUERYNO</td>
<td>INTEGER</td>
<td>No</td>
<td>No</td>
<td>Numeric identifier for explained SQL statement. For dynamic SQL statements (excluding the EXPLAIN SQL statement) issued through CLP or CLI, the default value is a sequentially incremented value. Otherwise, the default value is the value of STMTNO for static SQL statements and 1 for dynamic SQL statements.</td>
</tr>
<tr>
<td>QUERYTAG</td>
<td>CHAR(20)</td>
<td>No</td>
<td>No</td>
<td>Identifier tag for each explained SQL statement. For dynamic SQL statements issued through CLP (excluding the EXPLAIN SQL statement), the default value is 'CLP'. For dynamic SQL statements issued through CLI (excluding the EXPLAIN SQL statement), the default value is 'CLI'. Otherwise, the default value used is blanks.</td>
</tr>
<tr>
<td>STATEMENT_TYPE</td>
<td>CHAR(2)</td>
<td>No</td>
<td>No</td>
<td>Descriptive label for type of query being explained. Possible values are: CL (Call), CP (Compound SQL (Dynamic)), D (Delete), DC (Delete where current of cursor), I (Insert), M (Merge), S (Select), SI (Set Integrity or Refresh Table), U (Update), UC (Update where current of cursor)</td>
</tr>
<tr>
<td>UPDATABLE</td>
<td>CHAR(1)</td>
<td>No</td>
<td>No</td>
<td>Indicates if this statement is considered updatable. This is particularly relevant to SELECT statements which may be determined to be potentially updatable. Possible values are: ' ' (Not applicable (blank)), N (No), Y (Yes)</td>
</tr>
<tr>
<td>DELETABLE</td>
<td>CHAR(1)</td>
<td>No</td>
<td>No</td>
<td>Indicates if this statement is considered deletable. This is particularly relevant to SELECT statements which may be determined to be potentially deletable. Possible values are: ' ' (Not applicable (blank)), N (No), Y (Yes)</td>
</tr>
</tbody>
</table>
Table 257. EXPLAIN_STATEMENT Table (continued). PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable?</th>
<th>Key?</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TOTAL_COST</td>
<td>DOUBLE</td>
<td>No</td>
<td>No</td>
<td>Estimated total cost (in timerons) of executing the chosen access plan for this statement; set to 0 (zero) if EXPLAIN_LEVEL is 0 or E (original or effective text) since no access plan has been chosen at this time.</td>
</tr>
<tr>
<td>STATEMENT_TEXT</td>
<td>CLOB(2M)</td>
<td>No</td>
<td>No</td>
<td>Text or portion of the text of the SQL statement being explained. The text shown for the Plan Selection or Section Explain levels of Explain has been reconstructed from the internal representation and is SQL-like in nature; that is, the reconstructed statement is not guaranteed to follow correct SQL syntax.</td>
</tr>
<tr>
<td>SNAPSHOT</td>
<td>BLOB(10M)</td>
<td>Yes</td>
<td>No</td>
<td>Snapshot of internal representation for this SQL statement at the Explain_Level shown. This column is intended for use with DB2 Visual Explain. Column is set to NULL if EXPLAIN_LEVEL is not P (Plan Selection) since no access plan has been chosen at the time that this specific version of the statement is captured.</td>
</tr>
<tr>
<td>QUERY_DEGREE</td>
<td>INTEGER</td>
<td>No</td>
<td>No</td>
<td>Indicates the degree of intra-partition parallelism at the time of Explain invocation. For the original statement, this contains the directed degree of intra-partition parallelism. Otherwise, this contains the degree of intra-partition parallelism generated for the plan to use.</td>
</tr>
</tbody>
</table>
EXPLAIN_STREAM table

EXPLAIN_STREAM table

The EXPLAIN_STREAM table represents the input and output data streams between individual operators and data objects. The data objects themselves are represented in the EXPLAIN_OBJECT table. The operators involved in a data stream are to be found in the EXPLAIN_OPERATOR table.

Table 258. EXPLAIN_STREAM Table. PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable?</th>
<th>Key?</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXPLAIN_REQUESTER</td>
<td>VARCHAR(128)</td>
<td>No</td>
<td>FK</td>
<td>Authorization ID of initiator of this Explain request.</td>
</tr>
<tr>
<td>EXPLAIN_TIME</td>
<td>TIMESTAMP</td>
<td>No</td>
<td>FK</td>
<td>Time of initiation for Explain request.</td>
</tr>
<tr>
<td>SOURCE_NAME</td>
<td>VARCHAR(128)</td>
<td>No</td>
<td>FK</td>
<td>Name of the package running when the dynamic statement was explained or name of the source file when the static SQL was explained.</td>
</tr>
<tr>
<td>SOURCE_SCHEMA</td>
<td>VARCHAR(128)</td>
<td>No</td>
<td>FK</td>
<td>Schema, or qualifier, of source of Explain request.</td>
</tr>
<tr>
<td>SOURCE_VERSION</td>
<td>VARCHAR(64)</td>
<td>No</td>
<td>FK</td>
<td>Version of the source of the Explain request.</td>
</tr>
<tr>
<td>EXPLAIN_LEVEL</td>
<td>CHAR(1)</td>
<td>No</td>
<td>FK</td>
<td>Level of Explain information for which this row is relevant.</td>
</tr>
<tr>
<td>STMTNO</td>
<td>INTEGER</td>
<td>No</td>
<td>FK</td>
<td>Statement number within package to which this explain information is related.</td>
</tr>
<tr>
<td>SECTNO</td>
<td>INTEGER</td>
<td>No</td>
<td>FK</td>
<td>Section number within package to which this explain information is related.</td>
</tr>
<tr>
<td>STREAM_ID</td>
<td>INTEGER</td>
<td>No</td>
<td>No</td>
<td>Unique ID for this data stream within the specified operator.</td>
</tr>
<tr>
<td>SOURCE_TYPE</td>
<td>CHAR(1)</td>
<td>No</td>
<td>No</td>
<td>Indicates the source of this data stream:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>O Operator</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>D Data Object</td>
</tr>
<tr>
<td>SOURCE_ID</td>
<td>SMALLINT</td>
<td>No</td>
<td>No</td>
<td>Unique ID for the operator within this query that is the source of this data stream. Set to -1 if SOURCE_TYPE is ‘D’.</td>
</tr>
<tr>
<td>TARGET_TYPE</td>
<td>CHAR(1)</td>
<td>No</td>
<td>No</td>
<td>Indicates the target of this data stream:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>O Operator</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>D Data Object</td>
</tr>
<tr>
<td>TARGET_ID</td>
<td>SMALLINT</td>
<td>No</td>
<td>No</td>
<td>Unique ID for the operator within this query that is the target of this data stream. Set to -1 if TARGET_TYPE is ‘D’.</td>
</tr>
<tr>
<td>OBJECT_SCHEMA</td>
<td>VARCHAR(128)</td>
<td>Yes</td>
<td>No</td>
<td>Schema to which the affected data object belongs. Set to null if both SOURCE_TYPE and TARGET_TYPE are ‘O’.</td>
</tr>
<tr>
<td>OBJECT_NAME</td>
<td>VARCHAR(128)</td>
<td>Yes</td>
<td>No</td>
<td>Name of the object that is the subject of data stream. Set to null if both SOURCE_TYPE and TARGET_TYPE are ‘O’.</td>
</tr>
<tr>
<td>STREAM_COUNT</td>
<td>DOUBLE</td>
<td>No</td>
<td>No</td>
<td>Estimated cardinality of data stream.</td>
</tr>
<tr>
<td>COLUMN_COUNT</td>
<td>SMALLINT</td>
<td>No</td>
<td>No</td>
<td>Number of columns in data stream.</td>
</tr>
<tr>
<td>PREDICATE_ID</td>
<td>INTEGER</td>
<td>No</td>
<td>No</td>
<td>If this stream is part of a subquery for a predicate, the predicate ID will be reflected here, otherwise the column is set to -1.</td>
</tr>
</tbody>
</table>
### Table 258. EXPLAIN_STREAM Table (continued)

PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable?</th>
<th>Key?</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>COLUMN_NAMES</td>
<td>CLOB(2M)</td>
<td>Yes</td>
<td>No</td>
<td>This column contains the names and ordering information of the columns involved in this stream. These names will be in the format of: NAME1(A)+NAME2(D)+NAME3+NAME4 Where (A) indicates a column in ascending order, (D) indicates a column in descending order, and no ordering information indicates that either the column is not ordered or ordering is not relevant.</td>
</tr>
<tr>
<td>PMID</td>
<td>SMALLINT</td>
<td>No</td>
<td>No</td>
<td>Distribution map ID.</td>
</tr>
<tr>
<td>SINGLE_NODE</td>
<td>CHAR(5)</td>
<td>Yes</td>
<td>No</td>
<td>Indicates whether this data stream is on a single or on multiple database partitions: MULT On multiple database partitions COOR On coordinator node HASH Directed using hashing RID Directed using the row ID FUNC Directed using a function (HASHEDVALUE() or DBPARTITIONNUM()) CORR Directed using a correlation value Numeric Directed to predetermined single node</td>
</tr>
<tr>
<td>PARTITION_COLUMNS</td>
<td>CLOB(2M)</td>
<td>Yes</td>
<td>No</td>
<td>List of the columns on which this data stream is distributed.</td>
</tr>
<tr>
<td>SEQUENCE_SIZES</td>
<td>CLOB(2M)</td>
<td>Yes</td>
<td>No</td>
<td>Lists the expected sequence size for XML columns, or shows &quot;NA&quot; (not applicable) for any non-XML columns in the data stream. Set to null if there is not at least one XML column in the data stream.</td>
</tr>
</tbody>
</table>
Appendix J. Explain register values

Following is a description of the interaction of the CURRENT EXPLAIN MODE and CURRENT EXPLAIN SNAPSHOT special register values, both with each other and with the PREP and BIND commands.

With dynamic SQL, the CURRENT EXPLAIN MODE and CURRENT EXPLAIN SNAPSHOT special register values interact as follows.

Table 259. Interaction of Explain Special Register Values (Dynamic SQL)

<table>
<thead>
<tr>
<th>EXPLAIN SNAPSHOT values</th>
<th>EXPLAIN MODE values</th>
<th>RECOMMEND INDEXES</th>
<th>EVALUATE INDEXES</th>
</tr>
</thead>
<tbody>
<tr>
<td>NO</td>
<td>NO</td>
<td>• Explain tables populated.</td>
<td>• Explain tables populated.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Results of query returned.</td>
<td>• Results of query returned.</td>
</tr>
<tr>
<td></td>
<td>YES</td>
<td>• Explain tables populated.</td>
<td>• Explain tables populated.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Results of query not returned (dynamic statements not executed).</td>
<td>• Results of query not returned (dynamic statements not executed).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Explain tables populated when a statement qualifies for reoptimization at execution time.</td>
<td>• Explain tables populated when a statement qualifies for reoptimization at execution time.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Results of query returned.</td>
<td>• Results of query returned.</td>
</tr>
<tr>
<td></td>
<td>YES</td>
<td>• Explain tables populated.</td>
<td>• Explain tables populated.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Explain tables populated when a statement qualifies for reoptimization at execution time.</td>
<td>• Explain tables populated when a statement qualifies for reoptimization at execution time.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Results of query returned.</td>
<td>• Results of query returned.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Explain tables populated when a statement qualifies for reoptimization at execution time.</td>
<td>• Explain tables populated when a statement qualifies for reoptimization at execution time.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Results of query returned.</td>
<td>• Results of query returned.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Indexes recommended.</td>
<td>• Indexes recommended.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Indexes evaluated.</td>
<td>• Indexes evaluated.</td>
</tr>
</tbody>
</table>
## Explain register values

Table 259. Interaction of Explain Special Register Values (Dynamic SQL) (continued)

<table>
<thead>
<tr>
<th>EXPLAIN SNAPSH OT values</th>
<th>EXPLAIN MODE values</th>
<th>RECOMMEND INDEXES</th>
<th>EVALUATE INDEXES</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>NO</strong></td>
<td><strong>YES</strong></td>
<td><strong>EXPLAIN</strong></td>
<td><strong>REOPT</strong></td>
</tr>
<tr>
<td>EXPLAIN</td>
<td>• Explain tables populated.</td>
<td>• Explain tables populated.</td>
<td>• Explain tables populated.</td>
</tr>
<tr>
<td></td>
<td>• Explain Snapshot taken.</td>
<td>• Explain Snapshot taken.</td>
<td>• Explain Snapshot taken.</td>
</tr>
<tr>
<td></td>
<td>• Results of query not returned. (dynamic statements not executed).</td>
<td>• Results of query not returned. (dynamic statements not executed).</td>
<td>• Results of query not returned. (dynamic statements not executed).</td>
</tr>
<tr>
<td>REOPT</td>
<td>• Explain tables populated.</td>
<td>• Explain tables populated when a statement qualifies for reoptimization at execution time.</td>
<td>• Explain tables populated when a statement qualifies for reoptimization at execution time.</td>
</tr>
<tr>
<td></td>
<td>• Explain Snapshot taken when a statement qualifies for reoptimization at execution time.</td>
<td>• Explain Snapshot taken when a statement qualifies for reoptimization at execution time.</td>
<td>• Explain Snapshot taken when a statement qualifies for reoptimization at execution time.</td>
</tr>
<tr>
<td></td>
<td>• Results of query not returned. (dynamic or incremental-bind statements not executed).</td>
<td>• Results of query not returned. (dynamic or incremental-bind statements not executed).</td>
<td>• Results of query not returned. (dynamic or incremental-bind statements not executed).</td>
</tr>
<tr>
<td></td>
<td>• Results of query returned.</td>
<td>• Results of query returned.</td>
<td>• Results of query returned.</td>
</tr>
<tr>
<td></td>
<td>• Indexes recommended.</td>
<td>• Indexes recommended.</td>
<td>• Indexes recommended.</td>
</tr>
<tr>
<td></td>
<td>• Indexes evaluated.</td>
<td>• Indexes evaluated.</td>
<td>• Indexes evaluated.</td>
</tr>
</tbody>
</table>

The CURRENT EXPLAIN MODE special register interacts with the EXPLAIN bind option in the following way for dynamic SQL.

---

1078  SQL Reference, Volume 1
## Explain register values

**Table 260. Interaction of EXPLAIN Bind Option and CURRENT EXPLAIN MODE**

<table>
<thead>
<tr>
<th>EXPLAIN MODE values</th>
<th>EXPLAIN Bind option values</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>NO</td>
<td>NO</td>
<td>YES</td>
<td>REOPT</td>
</tr>
<tr>
<td></td>
<td>- Results of query returned.</td>
<td>- Explain tables populated for static SQL.</td>
<td>- Explain tables populated for static SQL when statement qualifies for reoptimization at execution time.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Results of query returned.</td>
<td>- Explain tables populated for dynamic SQL when statement qualifies for reoptimization at execution time.</td>
</tr>
<tr>
<td>YES</td>
<td>- Explain tables populated for dynamic SQL.</td>
<td>- Explain tables populated for static SQL.</td>
<td>- Explain tables populated for static SQL when statement qualifies for reoptimization at execution time.</td>
</tr>
<tr>
<td></td>
<td>- Results of query returned.</td>
<td>- Explain tables populated for dynamic SQL.</td>
<td>- Explain tables populated for dynamic SQL when statement qualifies for reoptimization at execution time.</td>
</tr>
<tr>
<td>EXPLAIN</td>
<td>- Explain tables populated for dynamic SQL.</td>
<td>- Explain tables populated for static SQL.</td>
<td>- Explain tables populated for static SQL when statement qualifies for reoptimization at execution time.</td>
</tr>
<tr>
<td></td>
<td>- Results of query not returned (dynamic statements not executed).</td>
<td>- Explain tables populated for dynamic SQL.</td>
<td>- Explain tables populated for dynamic SQL when statement qualifies for reoptimization at execution time.</td>
</tr>
</tbody>
</table>
## Explain register values

### Table 260. Interaction of EXPLAIN Bind Option and CURRENT EXPLAIN MODE (continued)

<table>
<thead>
<tr>
<th>EXPLAIN MODE values</th>
<th>EXPLAIN Bind option values</th>
<th>NO</th>
<th>YES</th>
<th>REOPT</th>
<th>ALL</th>
</tr>
</thead>
<tbody>
<tr>
<td>REOPT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Explain tables populated for dynamic SQL when statement qualifies for reoptimization at execution time.</td>
<td>• Explain tables populated for dynamic SQL when statement qualifies for reoptimization at execution time.</td>
<td>• Explain tables populated for static SQL when statement qualifies for reoptimization at execution time.</td>
<td>• Explain tables populated for static SQL when statement qualifies for reoptimization at execution time.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Results of query returned.</td>
<td>• Results of query returned.</td>
<td>• Results of query returned.</td>
<td>• Results of query returned.</td>
</tr>
<tr>
<td>RECOMMEND INDEXES</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Explain tables populated for dynamic SQL.</td>
<td>• Explain tables populated for static SQL.</td>
<td>• Explain tables populated for static SQL.</td>
<td>• Explain tables populated for static SQL.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Results of query not returned (dynamic statements not executed).</td>
<td>• Results of query not returned (dynamic statements not executed).</td>
<td>• Results of query not returned (dynamic statements not executed).</td>
<td>• Results of query not returned (dynamic statements not executed).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Recommend indexes.</td>
<td>• Recommend indexes.</td>
<td>• Recommend indexes.</td>
<td>• Recommend indexes.</td>
</tr>
<tr>
<td>EVALUATE INDEXES</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Evaluate indexes.</td>
<td>• Evaluate indexes.</td>
<td>• Evaluate indexes.</td>
<td>• Evaluate indexes.</td>
</tr>
</tbody>
</table>
### Explain register values

The CURRENT EXPLAIN SNAPSHOT special register interacts with the EXPLSNAP bind option in the following way for dynamic SQL.

#### Table 261. Interaction of EXPLSNAP bind Option and CURRENT EXPLAIN SNAPSHOT

<table>
<thead>
<tr>
<th>EXPLAIN SNAPSHOT values</th>
<th>NO</th>
<th>YES</th>
<th>REOPT</th>
<th>ALL</th>
</tr>
</thead>
<tbody>
<tr>
<td>NO</td>
<td>• Results of query returned.</td>
<td>• Explain Snapshot taken for static SQL.</td>
<td>• Explain Snapshot taken for static SQL when statement qualifies for reoptimization at execution time.</td>
<td>• Explain Snapshot taken for static SQL.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Results of query returned.</td>
<td>• Explain Snapshot taken for dynamic SQL when statement qualifies for reoptimization at execution time.</td>
<td>• Results of query returned.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Explain Snapshot taken for dynamic SQL.</td>
<td>• Results of query returned.</td>
</tr>
<tr>
<td>YES</td>
<td>• Explain Snapshot taken for dynamic SQL.</td>
<td>• Explain Snapshot taken for static SQL.</td>
<td>• Explain Snapshot taken for static SQL when statement qualifies for reoptimization at execution time.</td>
<td>• Explain Snapshot taken for static SQL.</td>
</tr>
<tr>
<td></td>
<td>• Results of query returned.</td>
<td>• Explain Snapshot taken for dynamic SQL.</td>
<td>• Explain Snapshot taken for dynamic SQL when statement qualifies for reoptimization at execution time.</td>
<td>• Results of query returned.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Results of query returned.</td>
<td>• Results of query returned.</td>
</tr>
<tr>
<td>EXPLAIN</td>
<td>• Explain Snapshot taken for dynamic SQL.</td>
<td>• Explain Snapshot taken for static SQL.</td>
<td>• Explain Snapshot taken for static SQL when statement qualifies for reoptimization at execution time.</td>
<td>• Explain Snapshot taken for static SQL.</td>
</tr>
<tr>
<td></td>
<td>• Results of query not returned (dynamic statements not executed).</td>
<td>• Explain Snapshot taken for dynamic SQL.</td>
<td>• Explain Snapshot taken for dynamic SQL when statement qualifies for reoptimization at execution time.</td>
<td>• Results of query not returned (dynamic statements not executed).</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Results of query not returned (dynamic statements not executed).</td>
<td>• Results of query not returned (dynamic statements not executed).</td>
</tr>
</tbody>
</table>
### Explain register values

**Table 261. Interaction of EXPLSNAP bind Option and CURRENT EXPLAIN SNAPSHOT (continued)**

<table>
<thead>
<tr>
<th>EXPLAIN SNAPSHOT values</th>
<th>EXPLSNAP Bind option values</th>
<th>NO</th>
<th>YES</th>
<th>REOPT</th>
<th>ALL</th>
</tr>
</thead>
<tbody>
<tr>
<td>REOPT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Explain Snapshot taken for dynamic SQL when statement qualifies for reoptimization at execution time.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Results of query returned.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Explain Snapshot taken for static SQL when statement qualifies for reoptimization at execution time.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Results of query returned.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Explain Snapshot taken for static SQL when statement qualifies for reoptimization at execution time.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Results of query returned.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Explain Snapshot taken for dynamic SQL when statement qualifies for reoptimization at execution time.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Results of query returned.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Explain Snapshot taken for static SQL when statement qualifies for reoptimization at execution time.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Results of query returned.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Explain Snapshot taken for dynamic SQL when statement qualifies for reoptimization at execution time.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Results of query returned.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Appendix K. Exception tables

Exception tables are user-created tables that mimic the definition of the tables that are specified to be checked using the SET INTEGRITY statement with the IMMEDIATE CHECKED option. They are used to store copies of the rows that violate constraints in the tables being checked.

The exception tables that are used by the load utility are identical to the ones described here, and can therefore be reused during checking with the SET INTEGRITY statement.

Rules for creating an exception table

The rules for creating an exception table are as follows:

- If the table is protected by a security policy, the exception table must be protected by the same security policy.
- The first “n” columns of the exception table are the same as the columns of the table being checked. All column attributes, including name, data type, and length should be identical. For protected columns, the security label protecting the column must be the same in both tables.
- All of the columns of the exception table must be free of constraints and triggers. Constraints include referential integrity and check constraints, as well as unique index constraints that could cause errors on insert.
- The “(n+1)” column of the exception table is an optional TIMESTAMP column. This serves to identify successive invocations of checking by the SET INTEGRITY statement on the same table, if the rows within the exception table have not been deleted before issuing the SET INTEGRITY statement to check the data. The timestamp precision can be any value from 0 to 12 and the value assigned will be the result of CURRENT_TIMESTAMP special register.
- The “(n+2)” column should be of type CLOB(32K) or larger. This column is optional but recommended, and will be used to give the names of the constraints that the data within the row violates. If this column is not provided (as could be warranted if, for example, the original table had the maximum number of columns allowed), then only the row where the constraint violation was detected is copied.
- The exception table should be created with both “(n+1)” and “(n+2)” columns.
- There is no enforcement of any particular name for the above additional columns. However, the type specification must be exactly followed.
- No additional columns are allowed.
- If the original table has generated columns (including the IDENTITY property), the corresponding columns in the exception table should not specify the generated property.
- Users invoking the SET INTEGRITY statement to check data must hold the INSERT privilege on the exception tables.
- The exception table cannot be a data partitioned table, a range clustered table, or a detached table.
- The exception table cannot be a materialized query table or a staging table.
- The exception table cannot have any dependent refresh immediate materialized query tables or any dependent propagate immediate staging tables.
Exception tables

The information in the “message” column has the following structure:

Table 262. Exception Table Message Column Structure

<table>
<thead>
<tr>
<th>Field number</th>
<th>Contents</th>
<th>Size</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Number of constraint violations</td>
<td>5 bytes</td>
<td>Right justified padded with '0'</td>
</tr>
<tr>
<td>2</td>
<td>Type of first constraint violation</td>
<td>1 byte</td>
<td>'K' - Check Constraint violation</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>'F' - Foreign Key violation</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>'G' - Generated Column violation</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>'I' - Unique Index violation</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>'D' - Delete Cascade violation</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>'P' - Data Partitioning violation</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>'S' - Invalid Row Security Label</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>'L' - DB2 LBAC Write rules violation</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>'X' - XML values index violation</td>
</tr>
<tr>
<td>3</td>
<td>Length of constraint/column/index ID</td>
<td>5 bytes</td>
<td>Right justified padded with '0'</td>
</tr>
<tr>
<td>4</td>
<td>Constraint name/Column name/index ID</td>
<td>length from the previous field</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Separator</td>
<td>3 bytes</td>
<td>&lt;space&gt;&lt;colon&gt;&lt;space&gt;</td>
</tr>
<tr>
<td>6</td>
<td>Type of next constraint violation</td>
<td>1 byte</td>
<td>'K' - Check Constraint violation</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>'F' - Foreign Key violation</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>'G' - Generated Column violation</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>'I' - Unique Index violation</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>'D' - Delete Cascade violation</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>'P' - Data Partitioning violation</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>'S' - Invalid Row Security Label</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>'L' - DB2 LBAC Write rules violation</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>'X' - XML values index violation</td>
</tr>
<tr>
<td>7</td>
<td>Length of constraint/column/index ID</td>
<td>5 bytes</td>
<td>Right justified padded with '0'</td>
</tr>
<tr>
<td>8</td>
<td>Constraint name/Column name/index ID</td>
<td>length from the previous field</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>Repeat Field 5 through 8 for each violation</td>
</tr>
</tbody>
</table>

- a Unique index violations will not occur during checking using the SET INTEGRITY statement, unless it is after an attach operation. This will be reported, however, when running LOAD if the FOR EXCEPTION option is chosen. LOAD, on the other hand, will not report check constraint, generated column, foreign key, delete cascade, or data partitioning violations in the exception tables.
- b To retrieve the expression of a generated column from the catalog views, use a select statement. For example, if field 4 is MYSCHEMA.MYTABLE.GEN_1, then SELECT SUBSTR(TEXT, 1, 50) FROM SYSCAT.COLUMNS WHERE TABSCHEMA='MYSCHEMA' AND TABNAME='MYNAME' AND COLNAME='GEN_1'; will return the first fifty bytes of the expression, in the form "AS (<expression>)"
- c To retrieve an index ID from the catalog views, use a select statement. For example, if field 4 is 1234, then SELECT INDSHEMA, INDNAMEN FROM SYSCAT.INDEXES WHERE IID=1234.
- d For XML values index violations, the constraint name, column name, or index ID field identifies the XML column that had an integrity violation in one of its indexes. It does not identify the index that had the integrity violation. It identifies only the name of the XML column on which the index violation occurs. For example, the value 'X00006XTCOL2' in the message column indicates an index violation occurred in one of the indexes on the XTCOL2 column.

Handling rows in an exception table

The information in exception tables can be processed in various ways. Data can be corrected and rows re-inserted into the original tables.

If there are no INSERT triggers on the original table, transfer the corrected rows by issuing an INSERT statement with a subquery on the exception table.

If there are INSERT triggers, and you want to complete the load operation with the corrected rows from exception tables without firing the triggers:
Exception tables

- Design the INSERT triggers to be fired depending on the value in a column that has been defined explicitly for the purpose.
- Unload data from the exception tables and append it using the load utility. In this case, if you want to recheck the data, note that constraints checking is not confined to the appended rows.
- Save the trigger definition text from the relevant system catalog view. Then drop the INSERT trigger and use INSERT to transfer the corrected rows from the exception tables. Finally, recreate the trigger using the saved trigger definition.

No explicit provision is made to prevent the firing of triggers when inserting rows from exception tables.

Only one violation per row is reported for unique index violations.

If values with LONG VARCHAR, LONG VARGRAPHIC, or LOB data types are in the table, the values are not inserted into the exception table in the case of unique index violations.

**Querying exception tables**

The message column structure in an exception table is a concatenated list of constraint names, lengths, and delimiters, as described earlier. This information can be queried.

For example, to retrieve a list of all violations, repeating each row with only the constraint name, assume that the original table T1 had two columns, C1 and C2. Assume also, that the corresponding exception table, E1, has columns C1 and C2, corresponding to those in T1, as well as a message column, MSGCOL. The following query uses recursion to list one constraint name per row (repeating rows that have more than one violation):

```sql
WITH IV (C1, C2, MSGCOL, CONSTNAME, I, J) AS
    (SELECT C1, C2, MSGCOL,
         CHAR(SUBSTR(MSGCOL, 12,
                      INTEGER(DECIMAL(VARCHAR(SUBSTR(MSGCOL,7,5)),5,0)) ),
                     1,
                     15+INTEGER(DECIMAL(VARCHAR(SUBSTR(MSGCOL,7,5)),5,0)))
      FROM E1
    UNION ALL
    SELECT C1, C2, MSGCOL,
         CHAR(SUBSTR(MSGCOL, J+6,
                      INTEGER(DECIMAL(VARCHAR(SUBSTR(MSGCOL,J+1,5)),5,0)) ),
                     I+1,
                     J+9+INTEGER(DECIMAL(VARCHAR(SUBSTR(MSGCOL,J+1,5)),5,0)))
      FROM IV
     WHERE I < INTEGER(DECIMAL(VARCHAR(SUBSTR(MSGCOL,1,5)),5,0))
 ) SELECT C1, C2, CONSTNAME FROM IV;
```

To list all of the rows that violated a particular constraint, the previous query could be extended as follows:

```sql
WITH IV (C1, C2, MSGCOL, CONSTNAME, I, J) AS
    (SELECT C1, C2, MSGCOL,
         CHAR(SUBSTR(MSGCOL, 12,
                      INTEGER(DECIMAL(VARCHAR(SUBSTR(MSGCOL,7,5)),5,0)) ),
                     1,
                     15+INTEGER(DECIMAL(VARCHAR(SUBSTR(MSGCOL,7,5)),5,0)))
      FROM E1
    UNION ALL
    SELECT C1, C2, MSGCOL,
         CHAR(SUBSTR(MSGCOL, J+6,
                      INTEGER(DECIMAL(VARCHAR(SUBSTR(MSGCOL,J+1,5)),5,0)) ),
                     I+1,
                     J+9+INTEGER(DECIMAL(VARCHAR(SUBSTR(MSGCOL,J+1,5)),5,0)))
      FROM IV
     WHERE I < INTEGER(DECIMAL(VARCHAR(SUBSTR(MSGCOL,1,5)),5,0))
 ) SELECT C1, C2, CONSTNAME FROM IV;
```
Exception tables

\[
\text{INTEGER}(\text{DECIMAL}(\text{VARCHAR})),
\]

\[
\text{I+1,}
\]

\[
\text{J+9+\text{INTEGER}(\text{DECIMAL}(\text{VARCHAR}))}
\]

FROM IV
WHERE I < \text{INTEGER}(\text{DECIMAL}(\text{VARCHAR})),
)

SELECT C1, C2, CONSTNAME FROM IV WHERE CONSTNAME = 'constraintname';

The following query could be used to obtain all of the check constraint violations:
WITH IV (C1, C2, MSGCOL, CONSTNAME, CONSTTYPE, I, J) AS
(SELECT C1, C2, MSGCOL,
 CHAR(SUBSTR(MSGCOL, 12,
  \text{INTEGER}(\text{DECIMAL}(\text{VARCHAR})), 5, 0))),
  \text{CHAR(SUBSTR(MSGCOL, 6, 1))},
  1,
  \text{15+\text{INTEGER}(\text{DECIMAL}(\text{VARCHAR}))}
FROM E1
UNION ALL
SELECT C1, C2, MSGCOL,
     \text{CHAR(SUBSTR(MSGCOL, J+6, 1))},
     \text{\text{I+1,}}
     \text{\text{J+9+\text{INTEGER}(\text{DECIMAL}(\text{VARCHAR}))}}
FROM IV
WHERE I < \text{INTEGER}(\text{DECIMAL}(\text{VARCHAR})),
)

SELECT C1, C2, CONSTNAME FROM IV WHERE CONSTTYPE = 'K';
Appendix L. SQL statements allowed in routines

The following table indicates whether or not an SQL statement (specified in the first column) is allowed to execute in a routine with the specified SQL data access indication. If an executable SQL statement is encountered in a routine defined with NO SQL, SQLSTATE 38001 is returned. For other execution contexts, SQL statements that are not supported in any context return SQLSTATE 38003. For other SQL statements not allowed in a CONTAINS SQL context, SQLSTATE 38004 is returned. In a READS SQL DATA context, SQLSTATE 38002 is returned. During creation of an SQL routine, a statement that does not match the SQL data access indication will cause SQLSTATE 42985 to be returned.

If a statement invokes a routine, the effective SQL data access indication for the statement will be the greater of:
- The SQL data access indication of the statement from the following table.
- The SQL data access indication of the routine specified when the routine was created.

For example, the CALL statement has an SQL data access indication of CONTAINS SQL. However, if a stored procedure defined as READS SQL DATA is called, the effective SQL data access indication for the CALL statement is READS SQL DATA.

When a routine invokes an SQL statement, the effective SQL data access indication for the statement must not exceed the SQL data access indication declared for the routine. For example, a function defined as READS SQL DATA could not call a stored procedure defined as MODIFIES SQL DATA.

Table 263. SQL Statement and SQL Data Access Indication

<table>
<thead>
<tr>
<th>SQL Statement</th>
<th>NO SQL</th>
<th>CONTAINS SQL</th>
<th>READS SQL DATA</th>
<th>MODIFIES SQL DATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALTER...</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>AUDIT</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>BEGIN DECLARE SECTION</td>
<td>Y(1)</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>CALL</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>CLOSE</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>COMMENT ON</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>COMMIT</td>
<td>N</td>
<td>N(4)</td>
<td>N(4)</td>
<td>N(4)</td>
</tr>
<tr>
<td>COMPOUND SQL</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>CONNECT(2)</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>CREATE...</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>DECLARE CURSOR</td>
<td>Y(1)</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>DECLARE GLOBAL TEMPORARY TABLE</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>DELETE</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>DESCRIBE</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>DISCONNECT(2)</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>DROP ...</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
</tr>
</tbody>
</table>
### SQL statements allowed in routines

Table 263. SQL Statement and SQL Data Access Indication (continued)

<table>
<thead>
<tr>
<th>SQL Statement</th>
<th>NO SQL</th>
<th>CONTAINS SQL</th>
<th>READS SQL DATA</th>
<th>MODIFIES SQL DATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>END DECLARE SECTION</td>
<td>Y(1)</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>EXECUTE</td>
<td>N</td>
<td>Y(3)</td>
<td>Y(3)</td>
<td>Y</td>
</tr>
<tr>
<td>EXECUTE IMMEDIATE</td>
<td>N</td>
<td>Y(3)</td>
<td>Y(3)</td>
<td>Y</td>
</tr>
<tr>
<td>EXPLAIN</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>FETCH</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>FREE LOCATOR</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>FLUSH EVENT MONITOR</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>GRANT ...</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>INCLUDE</td>
<td>Y(1)</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>INSERT</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>LOCK TABLE</td>
<td>N</td>
<td>y</td>
<td>y</td>
<td>Y</td>
</tr>
<tr>
<td>MERGE</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>OPEN</td>
<td>N</td>
<td>N</td>
<td>Y(5)</td>
<td>Y</td>
</tr>
<tr>
<td>PREPARE</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>REFRESH TABLE</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>RELEASE CONNECTION(2)</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>RELEASE SAVEPOINT</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>RENAME TABLE</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>REVOKE ...</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>ROLLBACK</td>
<td>N</td>
<td>N(4)</td>
<td>N(4)</td>
<td>N(4)</td>
</tr>
<tr>
<td>ROLLBACK TO SAVEPOINT</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>SAVEPOINT</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>SELECT INTO</td>
<td>N</td>
<td>N</td>
<td>Y(5)</td>
<td>Y</td>
</tr>
<tr>
<td>SET CONNECTION(2)</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>SET INTEGRITY</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>SET special register</td>
<td>N</td>
<td>y</td>
<td>y</td>
<td>Y</td>
</tr>
<tr>
<td>SET variable</td>
<td>N</td>
<td>y(6)</td>
<td>y(5)</td>
<td>Y</td>
</tr>
<tr>
<td>TRANSFER OWNERSHIP</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>TRUNCATE</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>UPDATE</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>VALUES INTO</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>WHENEVER</td>
<td>Y(1)</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
</tbody>
</table>

**Note:**
1. Although the NO SQL option implies that no SQL statements can be specified, non-executable statements are not restricted.
2. Connection management statements are not allowed in any routine execution context.
3. It depends on the statement being executed. The statement specified for the EXECUTE statement must be a statement that is allowed in the context of the particular SQL access level in effect. For example, if the SQL access level READS SQL DATA is in effect, the statement cannot be INSERT, UPDATE, or DELETE.

4. The COMMIT statement and the ROLLBACK statement without the TO SAVEPOINT clause can be used in a stored procedure, but only if the stored procedure is called directly from an application, or indirectly through nested stored procedure calls from an application. (If any trigger, function, method, or atomic compound statement is in the call chain to the stored procedure, COMMIT or ROLLBACK of a unit of work is not allowed.)

5. If the SQL access level READS SQL DATA is in effect, no SQL data change statement can be embedded in the SELECT INTO statement, in the cursor referenced by the OPEN statement, or the right hand side expression of the SET variable statement.

6. If the SQL access level CONTAINS SQL is in effect, no scalar fullselect can be embedded in the right hand side expression of the SET variable statement.

SQL statements allowed in routines
Appendix M. CALL invoked from a compiled statement

Invokes a procedure stored at the location of a database. A procedure, for example, executes at the location of the database, and returns data to the client application.

Programs using the SQL CALL statement are designed to run in two parts, one on the client and the other on the server. The server procedure at the database runs within the same transaction as the client application. If the client application and procedure are on the same database partition, the stored procedure is executed locally.

Note: This form of the CALL statement is deprecated, and is only being provided for compatibility with previous versions of DB2.

Invocation

This form of the CALL statement can only be embedded in an application program that is precompiled with the CALL_RESOLUTION DEFERRED option. It cannot invoke a federated procedure. It cannot be used in triggers, SQL procedures, or any other non-application contexts. It is an executable statement that cannot be dynamically prepared. However, the procedure name can be specified through a host variable and this, coupled with the use of the USING DESCRIPTOR clause, allows both the procedure name and the parameter list to be provided at run time, which achieves an effect similar to that of a dynamically prepared statement.

Authorization

The privileges held by the authorization ID of the statement at run time must include at least one of the following:

- EXECUTE privilege on the package that is associated with the procedure;
  EXECUTE privilege on the procedure is not checked
- CONTROL privilege on the package that is associated with the procedure
- SYSADM or DBADM authority

Syntax

```
CALL procedure-name (host-variable)
```

Description

`procedure-name` or `host-variable`

Identifies the procedure to call. The procedure name may be specified either directly or within a host variable. The procedure identified must exist at the current server (SQLSTATE 42724).

If `procedure-name` is specified, it must be an ordinary identifier that is not greater than 254 bytes. Because this can only be an ordinary identifier, it cannot
CALL invoked from a compiled statement

contain blanks or special characters. The value is converted to uppercase. If it is necessary to use lowercase names, blanks, or special characters, the name must be specified via a host-variable.

If host-variable is specified, it must be a CHAR or VARCHAR variable with a length attribute that is not greater than 254 bytes, and it must not include an indicator variable. The value is not converted to uppercase. The character string must be left-justified.

The procedure name can take one of several forms:

**procedure-name**
- The name (with no extension) of the procedure to execute. The procedure that is invoked is determined as follows.
  1. The **procedure-name** is used to search the defined procedures (in SYSCAT.ROUTINES) for a matching procedure. A matching procedure is determined using the steps that follow.
    a. Find the procedures (ROUTINETYPE is 'P') from the catalog (SYSCAT.ROUTINES), where the ROUTINENAME matches the specified **procedure-name**, and the ROUTINESCHEMA is a schema name in the SQL path (CURRENT PATH special register). If the schema name is explicitly specified, the SQL path is ignored, and only procedures with the specified schema name are considered.
    b. Next, eliminate any of these procedures that do not have the same number of parameters as the number of arguments specified in the CALL statement.
    c. Chose the remaining procedure that is earliest in the SQL path.
   
   If a procedure is selected, DB2 will invoke the procedure defined by the external name.
  2. If no matching procedure was found, **procedure-name** is used both as the name of the procedure library, and the function name within that library. For example, if **procedure-name** is proclib, the DB2 server will load the procedure library named proclib and execute the function routine proclib() within that library.

   On UNIX systems, the default directory for procedure libraries is sqllib/function. The default directory for unfenced procedures is sqllib/function/unfenced.

   In Windows-based systems, the default directory for procedure libraries is sqllib\function. The default directory for unfenced procedures is sqllib\function\unfenced.

   If the library or function could not be found, an error is returned (SQLSTATE 42884).

**procedure-library!function-name**
- The exclamation character (!) acts as a delimiter between the library name and the function name of the procedure. For example, if proclib!func is specified, proclib is loaded into memory, and the function func from that library is executed. This allows multiple functions to be placed in the same procedure library.

   The procedure library is located in the directories or specified in the LIBPATH variable, as described in **procedure-name**.

**absolute-path!function-name**
- The absolute-path specifies the complete path to the stored procedure library.
CALL invoked from a compiled statement

On a UNIX system, for example, if /u/terry/proclib!func is specified, the procedure library proclib is obtained from the directory /u/terry, and the function func from that library is executed.

In all of these cases, the total length of the procedure name, including its implicit or explicit full path, must not be longer than 254 bytes.

(host-variable,...)
Each specification of host-variable is a parameter of the CALL statement. The nth parameter of the CALL corresponds to the nth parameter of the server’s procedure.

Each host-variable is assumed to be used for exchanging data in both directions between client and server. To avoid sending unnecessary data between client and server, the client application should provide an indicator variable with each parameter, and set the indicator to -1 if the parameter is not used to transmit data to the procedure. The procedure should set the indicator variable to -128 for any parameter that is not used to return data to the client application.

If the server is DB2 9.1 database server, the parameters must have matching data types in both the client and server program.

USING DESCRIPTOR descriptor-name
Identifies an SQLDA that must contain a valid description of host variables. The nth SQLVAR element corresponds to the nth parameter of the server’s procedure.

Before the CALL statement is processed, the application must set the following fields in the SQLDA:

- SQLN to indicate the number of SQLVAR occurrences provided in the SQLDA
- SQLDABC to indicate the number of bytes of storage allocated for the SQLDA
- SQLD to indicate the number of variables used in the SQLDA when processing the statement
- SQLVAR occurrences to indicate the attributes of the variables. The following fields of each Base SQLVAR element passed must be initialized:
  - SQLTYPE
  - SQLLEN
  - SQLDATA
  - SQLIND
- The following fields of each Secondary SQLVAR element passed must be initialized:
  - LEN.SQLLONGLEN
  - SQLDATALEN
  - SQLDATATYPE_NAME

The SQLDA is assumed to be used for exchanging data in both directions between client and server. To avoid sending unnecessary data between client and server, the client application should set the SQLIND field to -1 if the parameter is not used to transmit data to the procedure. The procedure should set the SQLIND field -128 for any parameter that is not used to return data to the client application.
CALL invoked from a compiled statement

Notes

- **Use of large object (LOB) data types:**
  If the client and server application needs to specify LOB data from an SQLDA, allocate double the number of SQLVAR entries.
  LOB data types have been supported by procedures since DB2 Version 2. The LOB data types are not supported by all down level clients or servers.

- **Retrieving the DB2_RETURN_STATUS from an SQL procedure:**
  If an SQL procedure successfully issues a RETURN statement with a status value, this value is returned in the first SQLERRD field of the SQLCA. If the CALL statement is issued in an SQL procedure, use the GET DIAGNOSTICS statement to retrieve the DB2_RETURN_STATUS value. The value is -1 if the SQLSTATE indicates an error.

- **Returning result sets from procedures:**
  If the client application program is written using CLI, result sets can be returned directly to the client application. The procedure indicates that a result set is to be returned by declaring a cursor on that result set, opening a cursor on the result set, and leaving the cursor open when exiting the procedure.
  At the end of a procedure:
  - For every cursor that has been left open, a result set is returned to the application.
  - If more than one cursor is left open, the result sets are returned in the order in which their cursors were opened.
  - Only unread rows are passed back. For example, if the result set of a cursor has 500 rows, and 150 of those rows have been read by the procedure at the time the procedure is terminated, rows 151 through 500 will be returned to the procedure.

- **Handling of special registers:**
  The settings of special registers for the caller are inherited by the procedure on invocation, and restored upon return to the caller. Special registers may be changed within a procedure, but these changes do not affect the caller. This is not true for legacy procedures (those defined with parameter style DB2DARI, or found in the default library), where the changes made to special registers in a procedure become the settings for the caller.

- **Compatibilities**
  There is a newer, preferred, form of the CALL statement that can be embedded in an application (by precompiling the application with the CALL_RESOLUTION IMMEDIATE option), or that can be dynamically prepared.

Examples

**Example 1:**

In C, invoke a procedure called TEAMWINS in the ACHIEVE library, passing it a parameter stored in the host variable HV_ARGUMENT.

```c
strcpy(HV_PROCNAME, "ACHIEVE!TEAMWINS");
CALL :HV_PROCNAME (:HV_ARGUMENT);
```

**Example 2:**

In C, invoke a procedure called :SALARY_PROC, using the SQLDA named INOUT_SQLDA.
CALL invoked from a compiled statement

```c
struct sqlda *INOUT_SQLDA;
/* Setup code for SQLDA variables goes here */
CALL :SALARY_PROC
USING DESCRIPTOR :*INOUT_SQLDA;
```

**Example 3:**

A Java procedure is defined in the database, using the following statement:

```sql
CREATE PROCEDURE PARTS_ON_HAND ( IN PARTNUM INTEGER,
                                   OUT COST DECIMAL(7,2),
                                   OUT QUANTITY INTEGER )
    EXTERNAL NAME 'parts!onhand'
    LANGUAGE JAVA
    PARAMETER STYLE DB2GENERAL;
```

A Java application calls this procedure using the following code fragment:

```java
CallableStatement stpCall;
String sql = "CALL PARTS_ON_HAND (?, ?, ?)";
stpCall = con.prepareCall( sql ); /\ con is the connection */
stpCall.setInt( 1, variable1 ) ;
stpCall.setBigDecimal( 2, variable2 ) ;
stpCall.setInt( 3, variable3 ) ;
stpCall.registerOutParameter( 2, Types.DECIMAL,2 );
stpCall.registerOutParameter( 3, Types.INTEGER ) ;
stpCall.execute() ;
variable2 = stpCall.getBigDecimal(2) ;
variable3 = stpCall.getInt(3) ;
...
```

This application code fragment will invoke the Java method `onhand` in class `parts`, because the procedure name specified on the CALL statement is found in the database and has the external name `parts!onhand`.

Appendix M. CALL invoked from a compiled statement 1095
Appendix N. Overview of the DB2 technical information

DB2 technical information is available through the following tools and methods:

- DB2 Information Center
  - Topics (Task, concept and reference topics)
  - Help for DB2 tools
  - Sample programs
  - Tutorials
- DB2 books
  - PDF files (downloadable)
  - PDF files (from the DB2 PDF DVD)
  - printed books
- Command line help
  - Command help
  - Message help

Note: The DB2 Information Center topics are updated more frequently than either the PDF or the hardcopy books. To get the most current information, install the documentation updates as they become available, or refer to the DB2 Information Center at ibm.com.

You can access additional DB2 technical information such as technotes, white papers, and IBM Redbooks® publications online at ibm.com. Access the DB2 Information Management software library site at http://www.ibm.com/software/data/sw-library/.

Documentation feedback

We value your feedback on the DB2 documentation. If you have suggestions for how to improve the DB2 documentation, send an e-mail to db2docs@ca.ibm.com. The DB2 documentation team reads all of your feedback, but cannot respond to you directly. Provide specific examples wherever possible so that we can better understand your concerns. If you are providing feedback on a specific topic or help file, include the topic title and URL.

Do not use this e-mail address to contact DB2 Customer Support. If you have a DB2 technical issue that the documentation does not resolve, contact your local IBM service center for assistance.

Although the tables identify books available in print, the books might not be available in your country or region.

The form number increases each time a manual is updated. Ensure that you are reading the most recent version of the manuals, as listed below.

**Note:** The DB2 Information Center is updated more frequently than either the PDF or the hard-copy books.

### Table 264. DB2 technical information

<table>
<thead>
<tr>
<th>Name</th>
<th>Form Number</th>
<th>Available in print</th>
<th>Last updated</th>
</tr>
</thead>
<tbody>
<tr>
<td>Administrative API Reference</td>
<td>SC27-2435-01</td>
<td>Yes</td>
<td>November, 2009</td>
</tr>
<tr>
<td>Administrative Routines and Views</td>
<td>SC27-2436-01</td>
<td>No</td>
<td>November, 2009</td>
</tr>
<tr>
<td>Call Level Interface Guide and Reference, Volume 1</td>
<td>SC27-2437-01</td>
<td>Yes</td>
<td>November, 2009</td>
</tr>
<tr>
<td>Call Level Interface Guide and Reference, Volume 2</td>
<td>SC27-2438-01</td>
<td>Yes</td>
<td>November, 2009</td>
</tr>
<tr>
<td>Command Reference</td>
<td>SC27-2439-01</td>
<td>Yes</td>
<td>November, 2009</td>
</tr>
<tr>
<td>Data Movement Utilities Guide and Reference</td>
<td>SC27-2440-00</td>
<td>Yes</td>
<td>August, 2009</td>
</tr>
<tr>
<td>Data Recovery and High Availability Guide and Reference</td>
<td>SC27-2441-01</td>
<td>Yes</td>
<td>November, 2009</td>
</tr>
<tr>
<td>Database Administration Concepts and Configuration Reference</td>
<td>SC27-2442-01</td>
<td>Yes</td>
<td>November, 2009</td>
</tr>
<tr>
<td>Database Monitoring Guide and Reference</td>
<td>SC27-2458-00</td>
<td>Yes</td>
<td>August, 2009</td>
</tr>
<tr>
<td>DB2 Text Search Guide</td>
<td>SC27-2459-01</td>
<td>Yes</td>
<td>November, 2009</td>
</tr>
<tr>
<td>Developing ADO.NET and OLE DB Applications</td>
<td>SC27-2444-00</td>
<td>Yes</td>
<td>August, 2009</td>
</tr>
<tr>
<td>Developing Embedded SQL Applications</td>
<td>SC27-2445-01</td>
<td>Yes</td>
<td>November, 2009</td>
</tr>
<tr>
<td>Developing Java Applications</td>
<td>SC27-2446-01</td>
<td>Yes</td>
<td>November, 2009</td>
</tr>
<tr>
<td>Developing Perl, PHP, Python, and Ruby on Rails Applications</td>
<td>SC27-2447-00</td>
<td>No</td>
<td>August, 2009</td>
</tr>
</tbody>
</table>
### Table 264. DB2 technical information (continued)

<table>
<thead>
<tr>
<th>Name</th>
<th>Form Number</th>
<th>Available in print</th>
<th>Last updated</th>
</tr>
</thead>
<tbody>
<tr>
<td>Developing User-defined Routines (SQL and External)</td>
<td>SC27-2448-01</td>
<td>Yes</td>
<td>November, 2009</td>
</tr>
<tr>
<td>Getting Started with Database Application Development</td>
<td>GI11-9410-01</td>
<td>Yes</td>
<td>November, 2009</td>
</tr>
<tr>
<td>Getting Started with DB2 Installation and Administration on Linux and Windows</td>
<td>GI11-9411-00</td>
<td>Yes</td>
<td>August, 2009</td>
</tr>
<tr>
<td>Globalization Guide</td>
<td>SC27-2449-00</td>
<td>Yes</td>
<td>August, 2009</td>
</tr>
<tr>
<td>Installing DB2 Servers</td>
<td>GC27-2455-01</td>
<td>Yes</td>
<td>November, 2009</td>
</tr>
<tr>
<td>Installing IBM Data Server Clients</td>
<td>GC27-2454-00</td>
<td>No</td>
<td>August, 2009</td>
</tr>
<tr>
<td>Message Reference Volume 1</td>
<td>SC27-2450-01</td>
<td>No</td>
<td>November, 2009</td>
</tr>
<tr>
<td>Message Reference Volume 2</td>
<td>SC27-2451-01</td>
<td>No</td>
<td>November, 2009</td>
</tr>
<tr>
<td>Partitioning and Clustering Guide</td>
<td>SC27-2453-01</td>
<td>Yes</td>
<td>November, 2009</td>
</tr>
<tr>
<td>pureXML Guide</td>
<td>SC27-2465-01</td>
<td>Yes</td>
<td>November, 2009</td>
</tr>
<tr>
<td>Query Patroller Administration and User’s Guide</td>
<td>SC27-2467-00</td>
<td>No</td>
<td>August, 2009</td>
</tr>
<tr>
<td>Spatial Extender and Geodetic Data Management Feature User’s Guide</td>
<td>SC27-2468-00</td>
<td>No</td>
<td>August, 2009</td>
</tr>
<tr>
<td>SQL Procedural Languages: Application Enablement and Support</td>
<td>SC27-2470-00</td>
<td>Yes</td>
<td>August, 2009</td>
</tr>
<tr>
<td>SQL Reference, Volume 1</td>
<td>SC27-2456-01</td>
<td>Yes</td>
<td>November, 2009</td>
</tr>
<tr>
<td>SQL Reference, Volume 2</td>
<td>SC27-2457-01</td>
<td>Yes</td>
<td>November, 2009</td>
</tr>
<tr>
<td>Troubleshooting and Tuning Database Performance</td>
<td>SC27-2461-01</td>
<td>Yes</td>
<td>November, 2009</td>
</tr>
<tr>
<td>Upgrading to DB2 Version 9.7</td>
<td>SC27-2452-01</td>
<td>Yes</td>
<td>November, 2009</td>
</tr>
<tr>
<td>Visual Explain Tutorial</td>
<td>SC27-2462-00</td>
<td>No</td>
<td>August, 2009</td>
</tr>
<tr>
<td>What’s New for DB2 Version 9.7</td>
<td>SC27-2463-01</td>
<td>Yes</td>
<td>November, 2009</td>
</tr>
<tr>
<td>Workload Manager Guide and Reference</td>
<td>SC27-2464-00</td>
<td>Yes</td>
<td>August, 2009</td>
</tr>
<tr>
<td>XQuery Reference</td>
<td>SC27-2466-01</td>
<td>No</td>
<td>November, 2009</td>
</tr>
</tbody>
</table>
Table 265. DB2 Connect-specific technical information

<table>
<thead>
<tr>
<th>Name</th>
<th>Form Number</th>
<th>Available in print</th>
<th>Last updated</th>
</tr>
</thead>
<tbody>
<tr>
<td>Installing and Configuring DB2 Connect Personal Edition</td>
<td>SC27-2432-01</td>
<td>Yes</td>
<td>November, 2009</td>
</tr>
<tr>
<td>Installing and Configuring DB2 Connect Servers</td>
<td>SC27-2433-01</td>
<td>Yes</td>
<td>November, 2009</td>
</tr>
</tbody>
</table>

Table 266. Information Integration technical information

<table>
<thead>
<tr>
<th>Name</th>
<th>Form Number</th>
<th>Available in print</th>
<th>Last updated</th>
</tr>
</thead>
<tbody>
<tr>
<td>Information Integration: ASNCLP Program Reference for Replication and Event Publishing</td>
<td>SC19-1018-04</td>
<td>Yes</td>
<td>August, 2009</td>
</tr>
<tr>
<td>Information Integration: Configuration Guide for Federated Data Sources</td>
<td>SC19-1034-02</td>
<td>No</td>
<td>August, 2009</td>
</tr>
<tr>
<td>Information Integration: SQL Replication Guide and Reference</td>
<td>SC19-1030-02</td>
<td>Yes</td>
<td>August, 2009</td>
</tr>
<tr>
<td>Information Integration: Introduction to Replication and Event Publishing</td>
<td>GC19-1028-02</td>
<td>Yes</td>
<td>August, 2009</td>
</tr>
</tbody>
</table>

Ordering printed DB2 books

If you require printed DB2 books, you can buy them online in many but not all countries or regions. You can always order printed DB2 books from your local IBM representative. Keep in mind that some softcopy books on the DB2 PDF Documentation DVD are unavailable in print. For example, neither volume of the DB2 Message Reference is available as a printed book.

Printed versions of many of the DB2 books available on the DB2 PDF Documentation DVD can be ordered for a fee from IBM. Depending on where you are placing your order from, you may be able to order books online, from the IBM Publications Center. If online ordering is not available in your country or region, you can always order printed DB2 books from your local IBM representative. Note that not all books on the DB2 PDF Documentation DVD are available in print.

Note: The most up-to-date and complete DB2 documentation is maintained in the DB2 Information Center at [http://publib.boulder.ibm.com/infocenter/db2luw/v9r7](http://publib.boulder.ibm.com/infocenter/db2luw/v9r7)

To order printed DB2 books:
Ordering printed DB2 books

- To find out whether you can order printed DB2 books online in your country or region, check the IBM Publications Center at [http://www.ibm.com/shop/publications/order](http://www.ibm.com/shop/publications/order). You must select a country, region, or language to access publication ordering information and then follow the ordering instructions for your location.

- To order printed DB2 books from your local IBM representative:
  1. Locate the contact information for your local representative from one of the following Web sites:
     - The IBM directory of world wide contacts at [www.ibm.com/planetwide](http://www.ibm.com/planetwide)
     - The IBM Publications Web site at [http://www.ibm.com/shop/publications/order](http://www.ibm.com/shop/publications/order). You will need to select your country, region, or language to access appropriate publications home page for your location. From this page, follow the “About this site” link.
  2. When you call, specify that you want to order a DB2 publication.
  3. Provide your representative with the titles and form numbers of the books that you want to order. For titles and form numbers, see “DB2 technical library in hardcopy or PDF format” on page 1098.

Displaying SQL state help from the command line processor

DB2 products return an SQLSTATE value for conditions that can be the result of an SQL statement. SQLSTATE help explains the meanings of SQL states and SQL state class codes.

To start SQL state help, open the command line processor and enter:

```
? sqlstate or ? class code
```

where sqlstate represents a valid five-digit SQL state and class code represents the first two digits of the SQL state.
For example, ? 08003 displays help for the 08003 SQL state, and ? 08 displays help for the 08 class code.

Accessing different versions of the DB2 Information Center

For DB2 Version 9.7 topics, the DB2 Information Center URL is [http://publib.boulder.ibm.com/infocenter/db2luw/v9r7/](http://publib.boulder.ibm.com/infocenter/db2luw/v9r7/)

For DB2 Version 9.5 topics, the DB2 Information Center URL is [http://publib.boulder.ibm.com/infocenter/db2luw/v9r5](http://publib.boulder.ibm.com/infocenter/db2luw/v9r5)

For DB2 Version 9.1 topics, the DB2 Information Center URL is [http://publib.boulder.ibm.com/infocenter/db2luw/v9/](http://publib.boulder.ibm.com/infocenter/db2luw/v9/)

For DB2 Version 8 topics, go to the DB2 Information Center URL at: [http://publib.boulder.ibm.com/infocenter/db2luw/v8/](http://publib.boulder.ibm.com/infocenter/db2luw/v8/)
Displaying topics in your preferred language in the DB2 Information Center

The DB2 Information Center attempts to display topics in the language specified in your browser preferences. If a topic has not been translated into your preferred language, the DB2 Information Center displays the topic in English.

• To display topics in your preferred language in the Internet Explorer browser:
  1. In Internet Explorer, click the Tools —> Internet Options —> Languages... button. The Language Preferences window opens.
  2. Ensure your preferred language is specified as the first entry in the list of languages.
     – To add a new language to the list, click the Add... button.

     Note: Adding a language does not guarantee that the computer has the fonts required to display the topics in the preferred language.
     – To move a language to the top of the list, select the language and click the Move Up button until the language is first in the list of languages.
  3. Refresh the page to display the DB2 Information Center in your preferred language.

• To display topics in your preferred language in a Firefox or Mozilla browser:
  1. Select the button in the Languages section of the Tools —> Options —> Advanced dialog. The Languages panel is displayed in the Preferences window.
  2. Ensure your preferred language is specified as the first entry in the list of languages.
     – To add a new language to the list, click the Add... button to select a language from the Add Languages window.
     – To move a language to the top of the list, select the language and click the Move Up button until the language is first in the list of languages.
  3. Refresh the page to display the DB2 Information Center in your preferred language.

On some browser and operating system combinations, you must also change the regional settings of your operating system to the locale and language of your choice.

Updating the DB2 Information Center installed on your computer or intranet server

A locally installed DB2 Information Center must be updated periodically.

Before you begin

A DB2 Version 9.7 Information Center must already be installed. For details, see the “Installing the DB2 Information Center using the DB2 Setup wizard” topic in Installing DB2 Servers. All prerequisites and restrictions that applied to installing the Information Center also apply to updating the Information Center.

About this task

An existing DB2 Information Center can be updated automatically or manually:
Updating the DB2 Information Center installed on your computer or intranet server

- Automatic updates - updates existing Information Center features and languages. An additional benefit of automatic updates is that the Information Center is unavailable for a minimal period of time during the update. In addition, automatic updates can be set to run as part of other batch jobs that run periodically.

- Manual updates - should be used when you want to add features or languages during the update process. For example, a local Information Center was originally installed with both English and French languages, and now you want to also install the German language; a manual update will install German, as well as, update the existing Information Center features and languages. However, a manual update requires you to manually stop, update, and restart the Information Center. The Information Center is unavailable during the entire update process.

Procedure

This topic details the process for automatic updates. For manual update instructions, see the “Manually updating the DB2 Information Center installed on your computer or intranet server” topic.

To automatically update the DB2 Information Center installed on your computer or intranet server:

1. On Linux operating systems,
   a. Navigate to the path where the Information Center is installed. By default, the DB2 Information Center is installed in the /opt/ibm/db2ic/V9.7 directory.
   b. Navigate from the installation directory to the doc/bin directory.
   c. Run the ic-update script:
      ic-update

2. On Windows operating systems,
   a. Open a command window.
   b. Navigate to the path where the Information Center is installed. By default, the DB2 Information Center is installed in the <Program Files>\IBM\DB2 Information Center\Version 9.7 directory, where <Program Files> represents the location of the Program Files directory.
   c. Navigate from the installation directory to the doc\bin directory.
   d. Run the ic-update.bat file:
      ic-update.bat

Results

The DB2 Information Center restarts automatically. If updates were available, the Information Center displays the new and updated topics. If Information Center updates were not available, a message is added to the log. The log file is located in doc\eclipse\configuration directory. The log file name is a randomly generated number. For example, 1239053440785.log.
Manually updating the DB2 Information Center installed on your computer or intranet server

If you have installed the DB2 Information Center locally, you can obtain and install documentation updates from IBM.

About this task

Updating your locally-installed DB2 Information Center manually requires that you:

1. Stop the DB2 Information Center on your computer, and restart the Information Center in stand-alone mode. Running the Information Center in stand-alone mode prevents other users on your network from accessing the Information Center, and allows you to apply updates. The Workstation version of the DB2 Information Center always runs in stand-alone mode.
2. Use the Update feature to see what updates are available. If there are updates that you must install, you can use the Update feature to obtain and install them.

Note: If your environment requires installing the DB2 Information Center updates on a machine that is not connected to the internet, mirror the update site to a local file system using a machine that is connected to the internet and has the DB2 Information Center installed. If many users on your network will be installing the documentation updates, you can reduce the time required for individuals to perform the updates by also mirroring the update site locally and creating a proxy for the update site.

If update packages are available, use the Update feature to get the packages. However, the Update feature is only available in stand-alone mode.

3. Stop the stand-alone Information Center, and restart the DB2 Information Center on your computer.

Note: On Windows 2008, Windows Vista (and higher), the commands listed later in this section must be run as an administrator. To open a command prompt or graphical tool with full administrator privileges, right-click the shortcut and then select Run as administrator.

Procedure

To update the DB2 Information Center installed on your computer or intranet server:

1. Stop the DB2 Information Center.
   - On Windows, click Start → Control Panel → Administrative Tools → Services. Then right-click DB2 Information Center service and select Stop.
   - On Linux, enter the following command:
     
     ```
     /etc/init.d/db2icdv97 stop
     ```

2. Start the Information Center in stand-alone mode.
   - On Windows:
     a. Open a command window.
     b. Navigate to the path where the Information Center is installed. By default, the DB2 Information Center is installed in the Program_Files\IBM\DB2 Information Center\Version 9.7 directory, where Program_Files represents the location of the Program Files directory.
     c. Navigate from the installation directory to the doc\bin directory.
     d. Run the help_start.bat file:
Manually updating the DB2 Information Center installed on your computer or intranet server

help_start.bat
• On Linux:
  a. Navigate to the path where the Information Center is installed. By default, the DB2 Information Center is installed in the /opt/ibm/db2ic/V9.7 directory.
  b. Navigate from the installation directory to the doc/bin directory.
  c. Run the help_start script:

help_start
The systems default Web browser opens to display the stand-alone Information Center.
3. Click the Update button ( ). (JavaScript™ must be enabled in your browser.) On the right panel of the Information Center, click Find Updates. A list of updates for existing documentation displays.
4. To initiate the installation process, check the selections you want to install, then click Install Updates.
5. After the installation process has completed, click Finish.
6. Stop the stand-alone Information Center:
  • On Windows, navigate to the installation directory’s doc\bin directory, and run the help_end.bat file:

help_end.bat

Note: The help_end batch file contains the commands required to safely stop the processes that were started with the help_start batch file. Do not use Ctrl-C or any other method to stop help_start.bat.
  • On Linux, navigate to the installation directory’s doc/bin directory, and run the help_end script:

help_end

Note: The help_end script contains the commands required to safely stop the processes that were started with the help_start script. Do not use any other method to stop the help_start script.
7. Restart the DB2 Information Center.
  • On Windows, click Start → Control Panel → Administrative Tools → Services. Then right-click DB2 Information Center service and select Start.
  • On Linux, enter the following command:

/etc/init.d/db2icdv97 start

Results
The updated DB2 Information Center displays the new and updated topics.

DB2 tutorials
The DB2 tutorials help you learn about various aspects of DB2 products. Lessons provide step-by-step instructions.

Before you begin
You can view the XHTML version of the tutorial from the Information Center at http://publib.boulder.ibm.com/infolcenter/db2help/
DB2 tutorials

Some lessons use sample data or code. See the tutorial for a description of any prerequisites for its specific tasks.

DB2 tutorials

To view the tutorial, click the title.

“pureXML®” in pureXML Guide
Set up a DB2 database to store XML data and to perform basic operations with the native XML data store.

“Visual Explain” in Visual Explain Tutorial
Analyze, optimize, and tune SQL statements for better performance using Visual Explain.

DB2 troubleshooting information

A wide variety of troubleshooting and problem determination information is available to assist you in using DB2 database products.

DB2 documentation
Troubleshooting information can be found in the DB2 Troubleshooting Guide or the Database fundamentals section of the DB2 Information Center. There you will find information about how to isolate and identify problems using DB2 diagnostic tools and utilities, solutions to some of the most common problems, and other advice on how to solve problems you might encounter with your DB2 database products.

DB2 Technical Support Web site
Refer to the DB2 Technical Support Web site if you are experiencing problems and want help finding possible causes and solutions. The Technical Support site has links to the latest DB2 publications, TechNotes, Authorized Program Analysis Reports (APARs or bug fixes), fix packs, and other resources. You can search through this knowledge base to find possible solutions to your problems.


Terms and Conditions

Permissions for the use of these publications is granted subject to the following terms and conditions.

Personal use: You may reproduce these Publications for your personal, noncommercial use provided that all proprietary notices are preserved. You may not distribute, display or make derivative work of these Publications, or any portion thereof, without the express consent of IBM.

Commercial use: You may reproduce, distribute and display these Publications solely within your enterprise provided that all proprietary notices are preserved. You may not make derivative works of these Publications, or reproduce, distribute or display these Publications or any portion thereof outside your enterprise, without the express consent of IBM.

Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either express or implied, to the Publications or any information, data, software or other intellectual property contained therein.
Terms and Conditions

IBM reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of the Publications is detrimental to its interest or, as determined by IBM, the above instructions are not being properly followed.

You may not download, export or re-export this information except in full compliance with all applicable laws and regulations, including all United States export laws and regulations.

IBM MAKES NO GUARANTEE ABOUT THE CONTENT OF THESE PUBLICATIONS. THE PUBLICATIONS ARE PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, NONINFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.
Appendix O. Notices

This information was developed for products and services offered in the U.S.A. Information about non-IBM products is based on information available at the time of first publication of this document and is subject to change.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user’s responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan, Ltd.
3-2-12, Roppongi, Minato-ku, Tokyo 106-8711 Japan

The following paragraph does not apply to the United Kingdom or any other country/region where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions; therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites.
sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information that has been exchanged, should contact:

IBM Canada Limited
Office of the Lab Director
8200 Warden Avenue
Markham, Ontario
L6G 1C7
CANADA

Such information may be available, subject to appropriate terms and conditions, including, in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems, and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements, or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility, or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM’s future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information may contain examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious, and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application
programs conforming to the application programming interface for the operating
platform for which the sample programs are written. These examples have not
been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or
imply reliability, serviceability, or function of these programs. The sample
programs are provided "AS IS", without warranty of any kind. IBM shall not be
liable for any damages arising out of your use of the sample programs.

Each copy or any portion of these sample programs or any derivative work must
include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp.
Sample Programs. © Copyright IBM Corp. _enter the year or years_. All rights
reserved.

**Trademarks**

IBM, the IBM logo, and ibm.com® are trademarks or registered trademarks of
International Business Machines Corp., registered in many jurisdictions worldwide.
Other product and service names might be trademarks of IBM or other companies.
A current list of IBM trademarks is available on the Web at "Copyright and

The following terms are trademarks or registered trademarks of other companies

- Linux is a registered trademark of Linus Torvalds in the United States, other
countries, or both.
- Java and all Java-based trademarks and logos are trademarks of Sun
Microsystems, Inc. in the United States, other countries, or both.
- UNIX is a registered trademark of The Open Group in the United States and
other countries.
- Intel®, Intel logo, Intel Inside®, Intel Inside logo, Intel® Centrino®, Intel Centrino
logo, Celeron®, Intel® Xeon®, Intel SpeedStep®, Itanium, and Pentium® are
trademarks or registered trademarks of Intel Corporation or its subsidiaries in
the United States and other countries.
- Microsoft, Windows, Windows NT®, and the Windows logo are trademarks of
Microsoft Corporation in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of
others.
Index

Special characters

functions
  scalar 555
  SUBSTRB 555
  SUBSTRB scalar function
    details 555

A
ABS scalar function 341
ABSVAL scalar function 341
access plans
  description 46
ACOS scalar function
  details 342
ADD_MONTHS scalar function 343
ADVISE_INDEX table 1038
ADVISE_INSTANCE table 1042
ADVISE_MQT table 1043
ADVISE_PARTITION table 1045
ADVISE_TABLE table 1047
ADVISE_WORKLOAD table 1048
aggregate functions
  ARRAY_AGG 316
  COUNT 321
details 314
MIN 328
TRIM_ARRAY 590
UNNEST 671
aliases
  chaining process 13
details 54
names 54
overview 13
TABLE_NAME function 560
TABLE_SCHEMA function 561
ALL clause
  quantified predicate 281
  SELECT statement 689
ALL option 729
ambiguous reference errors 54
anchored data types 138
AND truth table 277
ANY clause 281
append mode tables
  comparison with other table types 6
application design
  character conversions
    SQL statements 30
code points for special characters 30
double-byte character support (DBCS) 30
application processes
  details 19
application requesters 31
arithmetic
  adding values
    columns 333
    expressions 333
  AVG function 318
arithmetic (continued)
  CORRELATION function 320
  COVARIANCE function 324
decimal values from numeric expressions 397
finding maximum value 327
floating point values from string expressions 512
floating-point values from numeric expressions 409, 512
integer values
  returning from expressions 355, 446
operators 215
regression functions 329
small integer values
  returning from expressions 547
STDDEV function 332
VARIANCE function 334
array constructors 243
ARRAY element
  specification 242
ARRAY_AGG function 316
ARRAY_DELETE scalar function 345
ARRAY_EXISTS predicate 284
ARRAY_FIRST scalar function 346
ARRAY_LAST scalar function 347
ARRAY_NEXT scalar function 348
ARRAY_PRIOR scalar function 349
arrays
  values 99
AS clause
  ORDER BY clause 689
  SELECT clause 689
ASCII scalar function
  details 350
ASIN scalar function
  details 351
assignments
  basic SQL operations 114
ATAN scalar function
details 352
ATAN2 scalar function
details 353
ATANH scalar function 354
attributes
  names 54
authorities
  overview 13
authorization IDs
details 54
authorization names
details 54
restrictions 54
AVG aggregate function 318

B
base tables
  comparison with other table types 6
BASE_TABLE function 669
basic predicate 280
BETWEEN predicate 285
BIGINT data type
  overview 82
BIGINT data type (continued)
precision 82
sign 82
BIGINT function 355
binary large objects (BLOBs)
definition 90
scalar functions 359
binary string data types 90
binding
function semantics 212
method semantics 212
bit data 85
bit manipulation functions 357
BITAND function 357
BITANDNOT function 357
BITNOT function 357
BITOR function 357
BITXOR function 357
BLOB data type
details 90
scalar functions 359
books
ordering 1100
buffer pools
names 54
built-in functions
details 190
string units 85
byte length
data type values 456

catalog views (continued)
COLUSE 801
CONDITIONS 802
CONSTDEP 803
CONTEXTATTRIBUTES 804
CONTEXTS 805
DATAPARTITIONEXPRESSION 806
DATAPARTITIONS 807
DATATYPEDEP 809
DATATYPES 810
DBAUTH 813
DBPARTITIONGROUPDEP 815
DBPARTITIONGROUPS 816
details 19
EVENTMONITORS 817
EVENTS 819
EVENTTABLES 820
FULLHIERARCHIES 821
FUNCMAPOPTIONS 822
FUNCMAPPARMOPTIONS 823
FUNCMAPPINGS 824
HIERARCHIES 825
HISTOGRAMTEMPLATEBINS 826
HISTOGRAMTEMPLATES 827
HISTOGRAMTEMPLATEUSE 828
INDEXAUTH 829
INDEXCOLUSE 830
INDEXDEF 831
INDEXES 832, 968
INDEXEXPLOITRULES 838
INDEXEXTENSIONDEP 839
INDEXEXTENSIONMETHODS 840
INDEXEXTENSIONPARMS 841
INDEXEXTENSIONS 842
INDEXOPTIONS 843
INDEXPARTITIONS 844
INDEXXMLPATTERNS 847
INVALIDOBJECTS 848
KEYCOLUSE 849
MODULEAUTH 850
MODULEOBJECTS 851
MODULES 852
NAMEMAPPINGS 853
NICKNAMES 854
overview 771, 773
PACKAGEAUTH 857
PACKAGEDEP 858
PACKAGES 860
PARTITIONMAPS 865
PASSTHRUAUTH 866
PREDICATESPECS 867
read-only 771
REFERENCES 868
ROLEAUTH 869
ROLES 870
ROUTINEAUTH 871
ROUTINENODEP 873
ROUTINEOPTIONS 875
ROUTINEPARTITIONS 876
ROUTINEPARMS 877
ROUTINES 880, 972
ROWFIELDS 889
SCHEMAAUTH 890
SCHEMATA 891
SECURITYLABELACCESS 892
SECURITYLABELCOMPONENTELEMENTS 893
CURRENT SCHEMA special register
details 177
CURRENT SERVER special register 178
CURRENT SQL_CCFLAGS special register 179
CURRENT SQLID special register 177
CURRENT TIME special register 180
CURRENT TIMESTAMP special register 181
CURRENT USER special register 183
cursor data types
casting 106
cursor predicates
details 286
cursor stability (CS)
details 21
cursor variables
names 54
cursors
names
defining 54
data
mixed
LIKE predicate 291
overview 85
data partitioning 33
data source objects
description 40
data sources 37
description 36
identifying 54
valid server types 976
data structures
packed decimal 761
data type mappings
description 41
forward 978
reverse 992
data types
anchored
overview 100
resolving anchor object 138, 140
array 99
BIGINT 82
binary string 90
BLOB 90
BOOLEAN
overview 96
CHAR 85
character string 85
CLOB 85
cursor
values 97
DATE 92
datetime 92
DBCLOB 89
DECIMAL
overview 82
determining for untyped expressions 265
DOUBLE 82
floating-point
overview 82
graphic string 89
INTEGER
overview 82

D
data
mixed
LIKE predicate 291
overview 85
data sources 37
description 36
identifying 54
valid server types 976
data structures
packed decimal 761
data type mappings
description 41
forward 978
reverse 992
data types
anchored
overview 100
resolving anchor object 138, 140
array 99
BIGINT 82
binary string 90
BLOB 90
BOOLEAN
overview 96
CHAR 85
character string 85
CLOB 85
cursor
values 97
DATE 92
datetime 92
DBCLOB 89
DECIMAL
overview 82
determining for untyped expressions 265
DOUBLE 82
floating-point
overview 82
graphic string 89
INTEGER
overview 82
data types (continued)
numeric
overview 82
partition compatibility 142
promotion 104
REAL 82
result columns 689
SMALLINT 82
SQL
overview 80
TIME 92
TIMESTAMP 92
TYPE_ID function 596
TYPE_NAME function 597
TYPE_SCHEMA function 598
unsupported 41
user-defined
overview 101
VARCHAR
overview 85
VARGRAPHIC 89
XML
values 98
XQuery
casting 106
database manager
limits 745
database partition compatibility
overview 142
databases
creating
SAMPLE 1003
distributed 1
partitioned 1
SAMPLE
dropping 1003
DATALINK data type
unsupported 41
DATAPARTITIONNUM scalar function 382
DATE data type
overview 92
WEEK_ISO scalar function 627
date data types
operations 226
DATE function 383
date functions
DAY 384
DAYS 389
MONTH 483
YEAR 668
dates
arithmetic 449, 489
string representation formats 92
DAY scalar function 384
DAYNAME scalar function
details 385
DAYOFWEEK scalar function
details 386
DAYOFWEEK_ISO scalar function
details 387
DAYOFYEAR scalar function
details 388
DAYS scalar function 389
DB2 Database for Linux, UNIX, and Windows data sources
default forward data type mappings 979
default reverse data type mappings 993

Index 1117
dynamic SQL
  SQLDA
details 761

E
EMPACT sample table 1003
EMPLOYEE sample table 1003
EMPHOTO sample table 1003
EMPRESUME sample table 1003
empty strings
  character 85
  graphic 89
EMPTY_BLOB scalar function 411
EMPTY_CLOB scalar function 411
EMPTY_DBCLOB scalar function 411
encoding
  schemes 28
ENCRYPT scalar function 412
encryption
  ENCRYPT function 412
  GETHINT function 423
  XMLGROUP function 337
  XMLROW function 653
error messages
  SQLCA definitions 755
ESCAPE clauses
  LIKE predicate 291
evaluation order
  expressions 215
event monitors
  EVENT_MON_STATE function 415
  names 54
  overview 32
Excel files
  supported versions 43
EXCEPT operator of fullselect 729
exception tables
  structure 1083
EXECUTE privilege
  functions 190
  methods 204
EXISTS predicate 288
EXP scalar function
details 416
explain tables
  overview 1037
EXPLAIN_ACTUALS table 1049
EXPLAIN_ARGUMENT table 1050
EXPLAIN_DIAGNOSTIC table
details 1058
EXPLAIN_DIAGNOSTIC_DATA table
details 1059
EXPLAIN_INSTANCE table 1060
EXPLAIN_OBJECT table 1063
EXPLAIN_OPERATOR table 1066
EXPLAIN_PREDICATE table 1068
EXPLAIN_STATEMENT table 1071
EXPLAIN_STREAM table 1074
exposed correlation names 54
expressions
  CASE 231
details 215
  field reference 239
  GROUP BY clause 689
  ORDER BY clause 689
  row 272
expressions (continued)
  ROW CHANGE 258
  SELECT clause 689
    structured type 264
    subselect 689
  external functions
    overview 190
  EXTRACT scalar function 417

F
federated database
  wrapper modules 37
  wrappers 37
federated databases
  description 37
  system catalog 45
federated server
  description 42
federated systems
  overview 35
field references
  row types 239
file reference variables
  BLOBs 54
  CLOBs 54
  DBCLOBs 54
fixed-length character string 85
fixed-length graphic string 89
flat files
  See also table-structured files 43
FLOAT data type
  precision 82
  sign 82
FLOAT function 419
floating-point constants 144
floating-point data types
  assignments 114
  conversion 114
FLOOR function
details 420
  SELECT statement 734
FOR FETCH ONLY clause
FOR READ ONLY clause
  SELECT statement 734
forward type mappings
  default mappings 978
  FROM clause
    identifiers 54
    subselect 689
fullselect
  detailed syntax 729
  examples 729
  initializing 734
  iterative 734
  multiple operations 729
  ORDER BY clause 689
  scalar 215
  subquery role 54
  table references 689
function mappings
  options 977
FUNCTION scalar function 381
functions
aggregate
  ARRAY_AGG 316
  COUNT 321
functions (continued)
aggregate (continued)
details 314
MIN 328
TRIM_ARRAY 590
UNNEST 671
XMLAGG 335
arguments 303
best fit 190
bit manipulation 357
built-in 190
casting
CAST 234
XMLCAST 240
column
ARRAY_AGG 316
AVG 318
CORR 320
CORRELATION 320
COUNT 321
COUNT_BIG 322
COVAR 324
COVARIANCE 324
MAX 327
MIN 328
overview 190
REGR_AVGX 329
REGR_AVGY 329
REGR_COUNT 329
REGR_INTERCEPT 329
REGR_R2 329
REGR_SLOPE 329
REGR_SXX 329
REGR_SXY 329
regression 329
STDDEV 332
SUM 333
TRIM_ARRAY 590
UNNEST 671
VAR 334
VARIANCE 334
XMLAGG 335
expressions 303
external
overview 190
invoking 190
mappings 54
names 54
OLAP 249
overloaded 190
overview 303
procedures 679
row 190
scalar
ABS 341
ABSVAL 341
ACOS 342
ADD_MONTHS 343
ARRAY_DELETE 345
ARRAY_FIRST 346
ARRAY_LAST 347
ARRAY_NEXT 348
ARRAY_PRIOR 349
ASCII 350
ASIN 351
functions (continued)
scalar (continued)
ATAN 352
ATAN2 353
ATANH 354
AVG 318
BIGINT 355
BITAND 357
BITANDNOT 357
BITNOT 357
BITOR 357
BITXOR 357
BLOB 359
CARDINALITY 360
CEIL 361
CEILING 361
CHAR 362
CHARACTER_LENGTH 368
CHR 370
CLOB 371
COALESCE 372
COLLATION_KEY_BIT 373
COMPARE_DECIMAL 375
CONCAT 377
COS 378
COSH 379
COT 380
DATE 383
DAY 384
DAYNAME 385
DAYOFWEEK 386
DAYOFWEEK_ISO 387
DAYOFYEAR 388
DAYS 389
DBCLOB 390
DBPARTITIONNUM 391
DEC 397
DECIMAL 397
DECIMAL_FORMAT 395
DECODE 401
DECRYPTBIN 403
DECRYPTCHAR 403
DEGREES 405
DEREF 406
DIFFERENCE 407
DIGITS 408
DOUBLE 409
DOUBLE_PRECISION 409
EMPTY_BLOB 411
EMPTY_CLOB 411
EMPTY_DBCLOB 411
ENCRYPT 412
EVENT_MON_STATE 415
EXP 416
EXTRACT 417
FLOAT 419
FLOOR 420
FUNCTION 381
GENERATE_UNIQUE 421
GETHINT 423
GRAPHIC 424
GREATEST 429
GROUPING 325
HASEDBVALUE 430
HEX 432
HOUR 434
functions (continued)
scalar (continued)
  IDENTITY_VAL_LOCAL 435
  INITCAP 439
  INSERT 441
  INSTR 445
  INT 446
  INTEGER 446
  JULIAN_DAY 448
  LAST_DAY 449
  LCASE 450
  LCASE (locale sensitive) 451
  LEAST 452
  LEFT 453
  LENGTH 456
  LN 458
  LOCATE 459
  LOCATE_IN_STRING 463
  LOG10 466
  LONGVARCHAR 467
  LONGVARCHAR (locale sensitive) 468
  LOWER 469
  LOWER (locale sensitive) 470
  LPAD 472
  LTRIM 475
  MAX 476
  MAX_CARDINALITY 477
  MICROSECOND 478
  MIDNIGHT_SECONDS 479
  MIN 480
  MINUTE 481
  MOD 482
  MONTH 483
  MONTHNAME 484
  MONTHS_BETWEEN 485
  MULTIPLY_ALT 487
  NEXT_DAY 489
  NODENUMBER (see functions, scalar, DBPARTITIONNUM) 391
  NORMALIZE_DECFLOAT 491
  NULLIF 492
  NVL 493
  OCTET_LENGTH 494
  OVERLAY 495
  overview 190, 339
  PARAMETER 499
  PARTITION (see functions, scalar, \n    HASHEDEVALUE) 430
  POSITION 500
  POSSTR 503
  POWER 505, 509
  QUANTIZE 506
  QUARTER 508
  RAISE_ERROR 510
  RAND 511
  REAL 512
  REC2XML 514
  REPEAT 518
  REPLACE 519
  RID 521
  RID_BIT 521
  RIGHT 523
  ROUND 526
  ROUND_TIMESTAMP 532
  RPAD 534
  RTRIM 537
  SECLABEL 538

functions (continued)
scalar (continued)
  SECLABEL_BY_NAME 539
  SECLABEL_TO_CHAR 540
  SECOND 542
  SIGN 544
  SIN 545
  SINH 546
  SMALLINT 547
  SOUNDEX 548
  SPACE 549
  SQRT 550
  STRIP 551
  SUBSTR 552
  SUBSTRING 558
  TABLE_NAME 560
  TABLE_SCHEMA 561
  TAN 563
  TANH 564
  TIME 565
  TIMESTAMP 566
  TIMESTAMP_FORMAT 568
  TIMESTAMPS 575
  TIMESTAMPDIFF 576
  TO_CHAR 578
  TO_CLOB 579
  TO_DATE 580
  TO_NUMBER 581
  TO_TIMESTAMP 582
  TOTALORDER 583
  TRANSLATE 585
  TRIM 588
  TRUNC 593
  TRUNC_TIMESTAMP 591
  TRUNCATE 593
  TYPE_ID 596
  TYPE_NAME 597
  TYPE_SCHEMA 598
  UCASE 599
  UCASE (locale sensitive) 600
  UPPER 601
  UPPER (locale sensitive) 602
  VALUE 604
  VARCHAR 605
  VARCHAR_FORMAT 611
  VARGRAPHIC 620
  WEEK 626
  WEEK_ISO 627
  XMLATTRIBUTES 628
  XMLCOMMENT 630
  XMLCONCAT 631
  XMLDOCUMENT 632
  XMLELEMENT 634
  XMLFOREST 641
  XMLGROUP 337
  XMLNAMESPACES 644
  XMLPARSE 646
  XMLPI 649
  XMLQUERY 650
  XMLROW 653
  XMLSERIALIZE 655
  XMLTEXT 657
  XMLVALIDATE 659
  XMLXSROBJECTID 663
  XSLTRANSFORM 664
  YEAR 668

signatures 190
L

label-based access control (LBAC)
exception tables 1083
limits 745
overview 13
security labels
component name length 745
name length 745
security policies
name length 745
labels
durations 226
object names in SQL procedures 54
large integers 82
large objects (LOBs)
details 91
locators
details 91
overview 91
partitioned tables 34
LAST_DAY scalar function 449
lateral correlation 689
LCASE (locale sensitive) scalar function
overview 450, 451
LEAST function 452
LEFT scalar function
details 453
LENGTH scalar function
details 456
LIKE predicate 291
limits
SQL 745
literals
details 144
LN function
details 458
local catalog
See global catalog 45
LOCATE scalar function
details 459
LOCATE_IN_STRING scalar function 463
locators
LOBs 91
variable details 54
locks
isolation levels 21
overview 19
LOG10 scalar function
details 466
logical operators
search rules 277
LONGVARCHAR function
details 467
LONGVARCHAR function
details 467
LOWER (locale sensitive) scalar function 470
LOWER scalar function 469
LPAD scalar function 472
LTRIM scalar function
details 475

M

MAX function 327, 476
MAX_CARDINALITY function 477
methods
best fit 204
built-in 204
dynamic dispatch 204
external 204
invoking 247
names 54
overloaded 204
signatures 204
SQL 204
type preserving 204
user-defined 204
MICROSECOND function 478
Microsoft Excel
See Excel files 43
Microsoft SQL Server
default forward type mappings 978
default reverse type mappings 992
supported versions 43
Microsoft SQL Server data sources
default forward data type mappings 985, 998
MIDNIGHT_SECONDS function 479
MIN aggregate function 328
MIN scalar function 480
MINUTE scalar function
details 481
MOD function
details 482
monitoring
database events
configuring event monitors 32
MONTH scalar function
details 483
MONTHNAME scalar function
details 484
months
date arithmetic 343, 485
MONTHS_BETWEEN scalar function 485
multibyte character support
code points for special characters 30
multidimensional clustering (MDC) tables
comparison with other table types 6
multiple row VALUES clause
result data type 130
MULTIPLY_ALT function 487

N

naming conventions
identifiers 54
qualified column rules 54
nested table expressions
subselect 689
NEXT_DAY scalar function 489
nickname column options
description 40
nicknames
description 40
FROM clause
exposed names 54
nonexposed names 54
subselect 689
qualifying column names 54
SELECT clause 689
NODENUMBER function 391
non-repeatable reads
isolation levels 21
nonexposed correlation-name in FROM clause 54
nonrelational data sources
   specifying data type mappings 41
NORMALIZE_DECFLOAT scalar function 491
NOT NULL clause
   NULL predicate 296
notices 1109
NUL-terminated character strings 85
NULL
   predicate rules 296
   SQL value
      assigning 114
      grouping-expressions 689
      occurrences in duplicate rows 689
      overview 80
      result columns 689
      specified by indicator variable 54
      unknown condition 277
NULLIF function 492
numbers
   precision 761
   scale 761
numeric assignments in SQL operations 114
numeric comparisons 114
NUMERIC data type
   precision 82
   sign 82
numeric data types
   summary 82
NVL scalar function 493
O
objects
   ownership 13
   tables 54
OCTET_LENGTH scalar function 494
ODBC
   core level functions 2
   DB2 CLI 2
   default forward type mappings 978
   supported versions 43
ODBC data sources
   default forward data type mappings 987
OLAP
   functions 249
   specification 249
OLE DB
   supported versions 43
operands
   decimal 215
   floating-point 215
   integer 215
   result data type 130
   strings 215
operations
   assignments 114
   comparisons 114
   datetime 226
   dereference 245
operators
   arithmetic 215
   optimizer
      description 46
   OR truth table 277
Oracle
   default forward type mappings 978
Oracle (continued)
   default reverse type mappings 992
Oracle NET8 data sources
   default forward data type mappings 988
   default reverse data type mappings 999
ORDER BY clause
   culturally correct collation 373
   SELECT statement 689
   order of evaluation 215
   ordering DB2 books 1100
   ordinary tokens 53
   ORG sample table 1003
   outer joins
   joined tables 689
   OVERLAY scalar function 495
   overloaded functions
      multiple function instances 190
   overloaded methods 204
   ownership
      database objects 13
P
packages
   authorization IDs
      binding 54
      dynamic statements 54
   names
      overview 54
      overview 13
   PARAMETER function 499
   parameter markers
      dynamic SQL
      host variables 54
      untyped 265
   parameters
      naming conventions 54
   PARTITION function 430
   partitioned database environments
      overview 33
      partition compatibility 142
   partitioned tables
      comparison with other table types 6
      large objects (LOBs) 34
paths
   SQL 190
   pattern matching
      Unicode databases 136
      performance
      isolation level effect 21
      phantom reads
      isolation levels 21
      points of consistency
      database 19
   POSITION scalar function 500
   POSSSTR function 503
   POWER scalar function
      details 505
      precedence
      overview 215
      precision
      numbers
      SQLLEN field 761
predicates
   ARRAY_EXISTS 284
   basic 280
   BETWEEN 285
predicates (continued)
cursor 273
EXISTS 288
IN 289
IS FOUND 286
IS NOT FOUND 286
IS NOT OPEN 286
IS OPEN 286
LIKE 291
NULL 296
overview 273
quantified 281
query processing 274
TYPE 297
VALIDATED 298
XML EXITS 300
privileges
EXECUTE
functions 190
methods 204
hierarchy 13
individual 13
overview 13
ownership 13
packages
implicit 13
problem determination
information available 1106
tutorials 1106
procedures
names
overview 54
PROJECT sample table 1003
pushdown analysis
description 46

Q
qualified names
reserved qualifiers 1031
uses 54
quantified predicates 281
QUANTIZE scalar function 506
QUARTER scalar function
details 508
queries
authorization IDs 687
examples
SELECT statement 734
fragments 46
overview 687
recursive 734
table expressions 2, 687
query optimization
description 46

R
RADIANS scalar function
details 509
RAISE ERROR scalar function 510
RAND scalar function
details 511
range-clustered tables
comparison with other table types 6
read stability (RS)
details 21
REAL function
details 512
single precision conversion 512
REAL SQL data type
precision 82
sign 82
REC2XML function 514
recursion queries 734
recursive common table expressions 734
reference types
casting 106
comparisons 114
DEREF function 406
details 101
regression functions
details 329
regular tables
comparison with other table types 6
remote authorization names 54
remote catalog information 45
remote function names 54
remote type names 54
REMOTE_NAME function mapping option
valid settings 977
remote-object-name 54
remote-schema-name 54
remote-table-name 54
REPEAT scalar function
details 518
repeatable read (RR)
details 21
REPLACE scalar function
details 519
remote-object name 54
remote-schema name 54
remote-table name 54
reverse type mappings
default mappings 992
RID function 521
RID.Bit function 521
RIGHT scalar function
details 523
rollbacks
overview 19
ROLLUP grouping of GROUP BY clause 689
ROUND scalar function
details 526
ROUND_TIMESTAMP scalar function 532
routines
procedures
overview 679
SQL statement support 1087
ROW anchored data type 100
ROW CHANGE expression 258
spaces
rules governing 53
special registers
  CLIENT ACCTNG 153
  CLIENT APPLNAME 154
  CURRENT_CLIENT_ACCTNG 153
  CURRENT_CLIENT_APPLNAME 154
  CURRENT_CLIENT_USERID 155
  CURRENT_CLIENT_WRKSTNNAME 156
  CURRENT DATE 157
  CURRENT DBPARTITIONNUM 158
  CURRENT DECFLOAT ROUNDDING MODE 159
  CURRENT DEFAULT TRANSFORM GROUP 160
  CURRENT DEGREE 161
  CURRENT EXPLAIN MODE 162
  CURRENT EXPLAIN SNAPSHOT 163
  CURRENT FEDERATED ASYNCHRONY 164
  CURRENT FUNCTION PATH 174
  CURRENT IMPLICIT XMLPARSE OPTION 165
  CURRENT ISOLATION 166
  CURRENT LOCALE LC_MESSAGES 167
  CURRENT LOCALE LC_TIME 168
  CURRENT LOCAL TIME 169
  CURRENT MAINTAINED TABLE TYPES FOR
    OPTIMIZATION 170
  CURRENT MDC ROLLOUT MODE 171
  CURRENT NODE (see special registers, CURRENT
    DBPARTITIONNUM) 158
  CURRENT OPTIMIZATION PROFILE 172
  CURRENT PACKAGE PATH 173
  CURRENT PATH 174
  CURRENT QUERY OPTIMIZATION 175
  CURRENT REFRESH AGE 176
  CURRENT SCHEMA 177
  CURRENT SERVER 178
  CURRENT SQL_CCFLAGS 179
  CURRENT SQLD 177
  CURRENT TIME 180
  CURRENT TIMESTAMP 181
  CURRENT TIMEZONE 182
  CURRENT USER 183
  Explain register values interactions 1077
  SESSION USER 184
  SQL language element 149
  SYSTEM USER 185
  updatable 149
  USER 186
specific names 54
specifications
  ARRAY element 242
  CAST 234
  OLAP 249
  XMLCAST 240
SQL
  assignments 114
  comparisons 114
  operations
    basic 114
    overview 1
    paths 190
    size limits 745
    variables
      names 54
SQL Access Group 2
SQL compiler
  in a federated system 37

SQL functions
  overview 190
SQL path
  overview 54
SQL statements
  allowed in routines 1087
  CALL 1091
  help
    displaying 1101
  names 54
SQL subqueries
  WHERE clause 689
SQL syntax
  AVG aggregate function 318
  basic predicate 280
  BETWEEN predicate 285
  comparing two predicates 280, 297
  CORRELATION aggregate function 320
  COUNT_BIG function 322
  COVARIANCE aggregate function 324
  EXISTS predicate 288
  GENERATE_UNIQUE function 421
  GROUP BY clause 689
  IN predicate 289
  LIKE predicate 291
  order of execution for multiple operations 729
  regression functions 329
  search conditions 277
  SELECT clause 689
  STDDEV aggregate function 332
  TYPE predicate 297
  VARIANCE aggregate function 334
  WHERE clause search conditions 689
SQLCA
  details 755
  error reporting 755
  partitioned database systems 755
  viewing interactively 755
SQLDA
  field in SQLDA 761
  contents 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  field in SQLDA 761
  function 510
  details 550
  STAFF sample table 1003
  STAFFG sample table 1003
  STDDEV function 332
  stored procedures
    CALL statement 1091
    XSR_ADDSCHEMADOC 679
    XSR_COMPLETE 680
    XSR_DTD 681
    XSR_EXTENDED 682
    XSR_REGISTER 684

Index 1127
stored procedures (continued)
  XSR_UPDATE 685
string units
  built-in functions 85
strings
  assignment conversion rules 114
  collating sequences 48
  conversion 28
  Unicode comparisons 136
STRIP scalar function
details 551
structured types
details 101
expressions 264
host variables
details 54
sub-total rows 689
subqueries 54
  HAVING clause 689
WHERE clause 689
subselect
details 689
SUBSTR scalar function
details 552
SUBSTRING scalar function
details 558
substrings
  SUBSTR function 552
SUM function 333
summary tables
comparison with other table types 6
super-aggregate rows 689
super-groups 689
supertypes
  identifier names 54
Sybase
default forward type mappings 978
default reverse type mappings 992
supported versions 43
Sybase data sources
default forward data type mappings 989
default reverse data type mappings 1000
symmetric super-aggregate rows 689
synonyms
  aliases 13
qualifying column names 54
syntax diagrams
  reading xi
system catalogs
  views
details 771
  overview 19
SYSTEM USER special register 185

T

TABLE clause
  subselect 689
table expressions
  common 734
  overview 2, 687
table functions
  details 190
  overview 668
table spaces
details 26
  names 54

TABLE_NAME function 560
TABLE_SCHEMA function 561
table-structured files
  supported versions 43
tables
  aliases 13
  append mode 6
base 6
catalog views on system tables 771
collation 33
correlation names 54
designator to avoid ambiguity 54
exception 1083
exposed names in FROM clause 54
FROM clause 689
multidimensional clustering (MDC) 6	names
details 54
FROM clause 689
nested table expressions 54
non-exposed names in FROM clause 54
overview 6
partitioned
  overview 6
qualified column names 54
range-clustered 6
reference 689
regular
  overview 6
result 6
SAMPLE database 1003
scalar fullselect 54
subqueries 54
summary 6
temporary
  overview 6
unique correlation names 54
TAN scalar function
details 563
TANH scalar function 564
temporary tables
comparison with other table types 6
Teradata
default forward type mappings 978
default reverse type mappings 992
Teradata data sources
default forward data type mappings 991, 1001
terms and conditions
  publications 1106
time
  expressions 565
  format conversion 362
  hour values in expressions 434
  returning
    microseconds from datetime value 478
    minutes from datetime value 481
    seconds from datetime value 542
    time stamp from values 566
    values based on time 565
  string representation formats 92
TIME data types
  operations 226
  overview 92
TIME functions 565
time stamps
  GENERATE_UNIQUE function 421
  rounding 532
timed stamps (continued)
   string representation formats 92
   truncating 591
TIMESTAMP data type
details 92
   WEEK scalar function 626
   WEEK_ISO scalar function 627
TIMESTAMP function 566
TIMESTAMP_FORMAT function 568
TIMESTAMP_ISO function 575
TIMESTAMPDIFF scalar function
details 576
   TO_CHAR function 578
   TO_CLOB scalar function 579
   TO_DATE function 580
   TO_NUMBER scalar function 581
   TO_TIMESTAMP scalar function 582
   tokens
   case sensitivity 53
   delimiter 53
   ordinary 53
   SQL language element 53
TOTALORDER scalar function 583
TRANSLATE scalar function 585
triggers
   cascading 10
   constraint interactions 1035
details 10
   Explain tables 1037
   interactions 1035
   maximum name length 745
   names 54
TRIM scalar function 588
TRIM_ARRAY function 590
troubleshooting
   online information 1106
tutorials 1106
TRUNC scalar function
details 593
TRUNC_TYPE_TIMESTAMP scalar function 591
TRUNCATE scalar function
details 593
truncation
   numbers 114
   truth tables 277
   truth valued logic 277
tutorials
   list 1105
   problem determination 1106
   troubleshooting 1106
   Visual Explain 1105
type names 54
TYPE predicate
   format 297
TYPE_ID function
data types 596
details 596
TYPE_NAME function
details 597
TYPE_SCHEMA function
data types 598
details 598
   type-mapping-name 54
   type-preserving methods 204
typed tables
   comparison with other table types 6
   names 54
   typed views
   names 54
   overview 11
U
   UCASE (locale sensitive) scalar function 600
   UCASE scalar function
details 599
   UDFs
   see user-defined functions (UDFs) 677
   unary operators
   minus sign 215
   plus sign 215
   uncommitted read (UR) isolation level
details 21
   undefined reference errors 54
   Unicode
   conventions 13
   conversion to uppercase 53
   Unicode UCS-2 encoding
   functions 339
   pattern matching 136
   string comparisons 136
   UNION operator
   role in comparison of fullselect 729
   unique correlation names
   table designators 54
   units of work (UOW)
   overview 19
   unknown condition
   null value 277
   UNNEST function 671
   unqualified names 54
   untyped expressions
determining data types 265
updates
   DB2 Information Center 1102, 1104
   updatable special registers 149
   UPPER (locale sensitive) scalar function 602
   UPPER scalar function 601
   user mappings
      description 39
      storing 39
   USER special register 186
   user-defined array types 99
   user-defined functions (UDFs)
details 190, 677
   overview 303
   user-defined methods
details 204
   user-defined types (UDTs)
casting 106
details 101
distinct types
details 101
   reference types 101
   structured types 101
   unsupported data types 41
V
   VALIDATED predicate 298
   VALUE function 604
values
   null 80