ACM Sigmod Record, Vol. 30, No. 1, March 2001

SQL and Management of External Data

Jim Melton
Oracle, Sandy, UT 84093
jim.melton@acm.org

Jan-Eike Michels
Vanja Josifovski
Krishna Kulkarni
Peter Schwarz
Kathy Zeidenstein
IBM, San Jose, CA
{janeike, vanja, krishnak, krzeide}@us.ibm.com
schwarz@almaden.ibm.com

Guest Column Introduction

In late 2000, work was completed on yet another part of the SQL standard [1], to which we introduced our readers in an earlier edition of this column [2].

Although SQL database systems manage an enormous amount of data, it certainly has no monopoly on that task. Tremendous amounts of data remain in ordinary operating system files, in network and hierarchical databases, and in other repositories. The need to query and manipulate that data alongside SQL data continues to grow. Database system vendors have developed many approaches to providing such integrated access.

In this (partly guested) article, SQL's new part, Management of External Data (SQL/MED), is explored to give readers a better notion of just how applications can use standard SQL to concurrently access their SQL data and their non-SQL data.

Jim Melton and Andrew Eisenberg

Managing External Data

The cost of writing applications that access SQL data using an SQL database management system and concurrently access non-SQL data using a different data manager continues to absorb resources unnecessarily. It is exacerbated by the necessity for application programmers to use different interfaces to access data under the control of different managers.

Responding to customer needs, Oracle, for example, offers its Open Transparent Gateway, Sybase has its OmniConnect, and IBM provides DataJoiner. Other vendors, including both major database vendors and smaller niche-market players, offer analogous products. More recently, IBM's *Garlic* research efforts [5] have focussed on providing an API to which third-party developers can build wrappers that allow SQL-servers to easily access a wide variety of non-SQL data sources.

SQL/MED addresses two aspects to the problem of accessing external data. The first aspect provides the ability to use the SQL interface to access non-SQL data (or even SQL data residing on a different database management system) and, if desired, to join that data with local SQL data. The application submits a single SQL query that references data from multiple sources to the SQL-server. That statement is then decomposed into fragments (or requests) that are submitted to the individual sources. The standard does not dictate how the query is decomposed, specifying only the interaction between the SQL-server and foreign-data wrapper that underlies the decomposition of the query and its subsequent execution. We will call this part of the standard the "wrapper interface"; it is described in the first half of this column.

The other aspect of the problem with external data is the *management* problem (as well as a retrieval problem). Huge amounts of critical data reside in file systems, including (at least from an SQL perspective) "non-traditional data", such as engineering diagrams, photographs, and other alternative media. There are often existing applications that rely on this data remaining in the file system, yet it is often advantageous to keep information *about* that file data in the database, because the database is easily queried. Problems with this very typical configuration include trying to keep database data and file data in sync, having to use a different interface for the files than for the database, and having to have different authorization mechanisms for file data and for database data. SQL/MED addresses these problems by introducing a new data type called `DATALINK`. The datalinks part of the standard is described in the latter half of the column.

Accessing External Data using the Wrapper Interface

SQL is based on the relational model, so external data must be represented as relational tables if it is to

fit seamlessly into the context of an SQL-server. SQL/MED introduces the notion of *foreign tables* to represent data stored externally to the SQL-server—an automobile price list can be represented as a foreign table with columns for make, model, year, price, *etc*. External data sources often make available several such collections, all of which can be accessed via a single network connection, so SQL/MED introduces the concept of a *foreign server* that allows access to a set of foreign tables—a single website may provide separate price lists for trucks, autos, motorbikes, *etc*., each presented as a separate table. When the SQL-server decomposes a query into fragments, each fragment is submitted to the appropriate foreign server for the foreign table referenced by that fragment. In the initial version of SQL/MED, a fragment cannot reference more than a single table. Future versions will allow query fragments referencing multiple foreign tables to be submitted to a foreign server as a single request.

It is also often true that several data sources share a common interface. In such cases, it is desirable to use a single code module to access all of these sources, each of which is represented by a foreign server. This common code module is manifested in SQL/MED by a *foreign-data wrapper*. Each foreign server to be accessed by a foreign-data wrapper can be characterized by configuration information—such as a host name and port number—that differentiates it from others accessed via the same wrapper. Because the kinds of information required are likely to vary from wrapper to wrapper, the standard does not specify a fixed set of configurable attributes. Instead, the concept of generic options—attribute/value pairs used by wrappers for configuration purposes—is introduced. Option values are cached in the catalogs of the SQL-server; the SQL-server does not interpret them, but makes their values available to the wrapper upon request. Generic options can be associated with each of the foreign data modeling concepts introduced by SQL/MED, including foreign servers, foreign tables, and their columns. Thus, a wrapper that represents data stored in files as foreign tables can associate an option with each table that specifies the character used to delimit fields in the corresponding file. The bulk of SQL/MED's text specifies an API — a set of functions — by which an SQL-server and a foreign-data wrapper conduct their business.

The SQL/MED functions that the SQL-server has to provide are called the *foreign-data wrapper interface SQL-server routines*. This term denotes all those functions that must be supported by a conforming SQL-server. By contrast, the *foreign-data wrapper interface wrapper routines* are the SQL/MED functions that a conforming foreign-data wrapper has to provide.

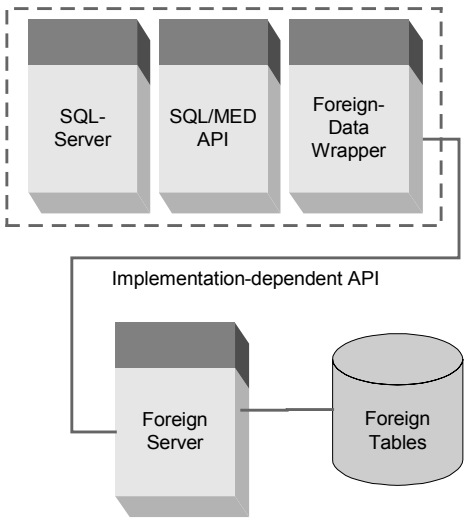

Figure 1 — Components Used by SQL/MED

In Figure 1, the relationships among these primary components defined and used by SQL/MED are illustrated. The SQL-server and the foreign-data wrapper communicate through the SQL/MED API, but they need not be implemented to run in a single process context; indeed, they may reside on separate computers connected by a network. Many configurations are possible, including one in which the foreign-data wrapper and the foreign server are combined into a single program. The communication between the foreign-data wrapper and the foreign server is determined solely by the authors of those components and is not addressed by SQL/MED.

The purpose of foreign tables is to support a transparent view on data — called external data — that is not stored and managed by the local SQL-server. By transparent, we mean that the user does not need to be aware of the fact that the data is not actually managed by the local SQL-server. Instead, the user can access a foreign table in a `SELECT` statement as though it were a regular base table or view. External data is presented to the user as ordinary SQL-data, even though it may be stored in a file system, in HTML-formatted web pages, or in some other specialized format.

Before the data of a foreign table can be accessed using SQL/MED's facilities, the SQL-server must first be informed of the foreign table's existence. This is done using the `CREATE FOREIGN TABLE` statement:

```
CREATE FOREIGN TABLE table-name
  [ ( col-def, col-def, ... ) ]
  SERVER foreign-server-name
  [ generic-options ]
```

Execution of this statement creates a new schema object, a foreign table (perhaps more clearly, a *for-*

eign table descriptor), in a schema that belongs to the SQL-server. The foreign table (descriptor) identifies the foreign server that manages the data to be presented as though it were a table.

Of course, since creation of a foreign table requires a reference to a foreign server, the foreign server (that is, its representation in the metadata of the SQL-server) must first be created:

```
CREATE SERVER server-name
  [ TYPE server-type ]
  [ VERSION server-version ]
  [ AUTHORIZATION auth-info ]
  FOREIGN DATA WRAPPER w-name
  [ generic-options ]
```

The optional `TYPE` and `VERSION` values are meaningful only to specific implementations and valid values are not specified by SQL/MED. The optional `AUTHORIZATION` clause allows the statement to specify the effective owner (from the viewpoint of the SQL-server) of the foreign server. Unlike virtually all other SQL objects, including foreign tables, foreign servers are not represented in a schema belonging to the SQL-server, but are represented at a higher level: the catalog (which contains schemas and a few other objects).

Of course, in order to create (the descriptor that represents) a foreign server, we have to name the foreign-data wrapper that manages the foreign server, so we must first execute:

```
CREATE FOREIGN DATA WRAPPER wrap-name
  [ AUTHORIZATION auth-id ]
  [ LIBRARY library-name ]
  LANGUAGE language-name
  [ generic-options ]
```

As with foreign servers, foreign-data wrappers' representations are stored in an SQL-server's catalogs and not in a schema. Similarly, the optional `AUTHORIZATION` clause allows specification of the nominal owner of the wrapper.

The optional `LIBRARY` clause specifies a character string literal that identifies a software library (using an implementation-defined syntax) containing those SQL/MED routines that implement the specific foreign-data wrapper. (Some SQL/MED routines are implemented by the SQL-server, while others are implemented as part of the foreign-data wrapper.) The `LANGUAGE` clause specifies the name of the programming language in which the library routines are written (C or PL/I, for example).

The sequence of statement execution must be:

```
CREATE FOREIGN DATA WRAPPER ...
CREATE FOREIGN SERVER ...
CREATE FOREIGN TABLE...
```

Alternatively, once the foreign server has been created, it *may* be possible (depending entirely on the capabilities of the foreign-data wrapper and the foreign server) to execute a single statement to create one or more foreign tables based on information available from the foreign server:

```
IMPORT FOREIGN SCHEMA schema-name
  [ LIMIT TO ( table-name-list )
    | EXCEPT ( table-name-list ) ]
  FROM SERVER server-name
  INTO local-schema-name
```

This statement presumes that the foreign server recognizes the concept of a schema containing tables (or that the foreign-data wrapper simulates that concept). Execution of this statement allows importation of the table definitions (including those tables' column definitions) for every table contained in the specified foreign schema — possibly limited to certain tables or with specified tables omitted.

In many cases, foreign servers recognize the concept of *ownership* of the data they manage. For those situations, the SQL-server is given the ability to map its user identifiers to those of the foreign server through the statement:

```
CREATE USER MAPPING FOR auth-id
  SERVER server-name
  [ generic-options ]
```

SQL statements involving foreign tables managed by the specified foreign server are executed as though the authorization identifier at the SQL-server were "really" the corresponding entity recognized by the foreign server.

Careful readers will observe that we have not yet discussed the various occurrences of `generic-options` that appear in most of those statements we've described. Generic options, introduced earlier in this paper, are specified using the syntax we've shown here.

Processing Queries with Foreign-Data Wrappers

Communication between an SQL-server and a foreign-data wrapper can occur in either of two modes: *decomposition mode* or *pass-through mode*. In pass-through mode, the SQL-server transfers the query string, as is, to the foreign-data wrapper. The wrapper and the data source are solely responsible for analyzing and executing the query. Although implementation of pass-through mode by a foreign-data wrapper is optional, it is especially useful when the foreign server is also an SQL-engine. Due to space constraints, we do not discuss pass-through mode further in this paper.

In decomposition mode, the SQL-server breaks a query into fragments, each to be executed by a particular foreign server. The interaction between the SQL-server and the foreign-data wrapper can be divided into two phases:

- A *query planning* phase, in which the foreign-data wrapper and the SQL-server cooperatively produce an execution plan for the fragment.
- A *query execution* phase, in which the agreed-upon plan is executed and foreign data is returned to the SQL-server.

In both the query planning and execution phases, information must be exchanged between the foreign-data wrapper and the SQL-server. To make conforming implementations simpler, and to facilitate implementation in different programming language environments, SQL/MED utilizes a functional interface based on handles. For example, suppose information managed by the SQL-server is to be passed to a foreign-data wrapper. Instead of explicitly defining the layout of a data structure for this purpose, the standard requires the SQL-server to pass an integer handle representing such a structure to the foreign-data wrapper. For each type of handle that can be exchanged, the standard specifies a set of applicable functions that take a handle as a parameter and extract a value from the corresponding data structure. To simplify the description that follows, we will not explicitly refer to handles. However, the reader should assume that whenever a data structure is passed from the SQL-server to a foreign-data wrapper or vice-versa, the exchange is done by means of a handle.

Query Planning Phase

During query planning, the interaction between the SQL-server and the foreign-data wrapper is based on a request/reply paradigm. The SQL-server builds a request representing the query fragment. The foreign-data wrapper analyzes the request and returns a reply that describes that portion of the request that can be handled by the foreign server. The SQL-server must compensate for any part of the query fragment that cannot be executed by the foreign server.

The flexibility of this paradigm is essential, since the query processing capabilities of data sources may vary widely. Experience in research and industry has shown that a declarative approach to describing the capabilities of data sources leads to an unmanageable explosion of descriptive attributes. As will be seen below, the full power of this paradigm is not exploited in the initial version of SQL/MED. However, we believe it will be essential for accommodating the wide array of data sources that should be accessible using SQL/MED.

Before execution planning for a query fragment can begin, the SQL-server must identify the relevant foreign server and create a *connection*. Since a foreign-data wrapper may simultaneously manage multiple connections to various foreign servers, the connection provides a context for subsequent interaction between the SQL-server and a particular foreign server. Creation of a connection does not imply that the foreign-data wrapper actually connects to the data source at this time; whether and when the foreign-data wrapper establishes a connection to the data source is up to the wrapper implementation. To create a connection to a foreign server, the SQL-server invokes the `ConnectServer()` routine provided by the appropriate foreign-data wrapper, supplying (via a handle) a data structure that identifies the foreign server and includes the names and values of any generic options that were supplied when the foreign server was defined by DDL. The SQL-server also supplies another data structure that describes the user on whose behalf the query is being executed and that contains the user mapping information described earlier. This information may be used by the foreign-data wrapper to authenticate the user at the foreign server. `ConnectServer()` returns the newly-established connection to the SQL-server. Once a connection has been created, it is not necessarily destroyed after processing a single query fragment. The SQL-server can preserve and reuse the connection later.

Once a connection is established, the SQL-server invokes the foreign-data wrapper's `InitRequest()` routine, passing the query fragment to the foreign-data wrapper in the form of an SQL/MED *request*. A request is a data structure that abstractly describes the SQL statement corresponding to the query fragment, rather than an explicit representation of the statement as a character string. This approach eliminates the need for each foreign-data wrapper to parse SQL, an onerous task for wrappers designed for data sources that do not use SQL. The components of a request are subordinate data structures representing the individual clauses of an SQL statement, *e.g.*, the `SELECT` clause, `FROM` clause, *etc.*

Each element of the `SELECT` list is represented by a *value expression* that describes a result column of the fragment. In the initial version of the standard, each value expression must denote a simple column name; future versions will support more complex expressions. Each element of the `FROM` clause is represented by a *table reference* that identifies a foreign table. The initial version of the standard limits the `FROM` clause to a single table reference. Other clauses, such as `WHERE`, `ORDER BY`, *etc.*, are also not currently supported. While future versions of SQL/MED will allow for more complex requests, the effect of the current limitations is that only query fragments of the form "`SELECT <column_list> FROM` *`FTN`*" can be described, where *FTN* is the name of a foreign table and each element of `<col-`

umn_list> refers to a column of that table. Since all the columns in the request are needed for the SQL-server to produce the complete query result, the foreign-data wrapper must be able to process the entire request.

However, once the standard supports more complex requests, a foreign-data wrapper may be unable to process the entire request. For example, the request might include a predicate that cannot be evaluated by the foreign server. In such a case, the foreign-data wrapper could return only the basic data values and the SQL-server could compensate by applying the predicate and filtering the result. When a foreign-data wrapper receives a request via the `InitRequest()` routine, it examines the request by invoking routines implemented by the SQL-server ("foreign-data wrapper interface SQL-server routines", in the parlance of the standard). Routines are provided to extract table references from the `FROM` clause (`GetTableRefElem()`, `GetTableRefTableName()`), as well as the values of generic options associated with a referenced table (`GetTableOpts()`). Other routines supply information about the columns in the SELECT list (`GetSelectElem()`, `GetValExprColName()`) and their generic option values (`GetTableColOpt()`).

Once the foreign-data wrapper has analyzed the request, it constructs an SQL/MED *reply*. The structure of a reply is similar to that of a request, but the corresponding routines for examining the reply (`GetReplyTableRef()`, `GetReplySelectElem()`, *etc.*) are implemented by the foreign-data wrapper (*i.e.*, they are "foreign-data wrapper interface wrapper routines"). The reply is returned to the SQL-server from `InitRequest()`, along with a second data structure, an *execution plan*. The content of this second data structure is determined solely by the foreign-data wrapper, and it is not interpreted by the SQL-server. Its purpose is to encapsulate all the information that is needed by the foreign-data wrapper to execute the portion of the query fragment represented by the reply. As will be described below, the SQL-server will hand the execution plan back to the foreign-data wrapper at the start of the query execution phase. The execution plan is the wrapper's means of preserving information between the planning and execution phases of query processing. By contrast, the SQL-server can discard the reply when query planning is complete.

Query Execution Phase

During the execution phase, the portion of the query fragment described by the reply is executed by the foreign-data wrapper and the underlying data source. To initiate query execution, the SQL-server invokes the `Open()` routine in the foreign-data wrapper, passing the execution plan as an argument. To fetch a row of the result, the SQL-server invokes the `Iterate()` routine. Once all rows have been fetched, the SQL-server invokes the `Close()` routine to allow the foreign-data wrapper to clean-up after the execution. The execution plan may be reused, for example if the query fragment represents the inner table of a nested-loop join. When it is no longer needed, the SQL-server invokes the `FreeExecutionHandle()` routine to deallocate the execution plan.

SQL/MED makes use of *descriptors* to exchange data between the SQL-server and a foreign-data wrapper. These descriptors are adapted from those used in the SQL/CLI standard [3]. Descriptors are implemented by the SQL-server, and encapsulate both the types and values of a row of data. Depending on the SQL-server implementation environment, the value may be stored in the descriptor itself, or the descriptor may point to a buffer. Use of pointers is generally more efficient, but they are difficult to support in some languages (*e.g.*, Java).

SQL/MED defines only the fields that make up the descriptors; it does not define the actual programming language data structures that implement them. As with other data structures we have described, SQL/MED specifies various routines that get and set the value of a descriptor field when given a descriptor and the field's identity.

A descriptor consists of zero or more *descriptor areas*. Each area describes one column, which can be an instance of any SQL data type, including types like `ROW` and `ARRAY` as well as basic types like `INTEGER`, `VARCHAR`, *etc.* If the implementation programming language of the wrapper supports a data type that corresponds directly to the SQL data type in the descriptor (as described in the SQL/CLI standard) the SQL-server must be able to convert this standard representation to any internal representation it requires. If no such correspondence exists (*e.g.*, `DATE` cannot be directly mapped into any programming language that SQL/MED supports), then the wrapper should use the canonical character string representation of the type as described in SQL/Foundation [4].

A Simple Example

Example 1 illustrates the principles outlined above.

Example 1: Consider a table of employees stored in a Unix® text file. Each line of the file contains one employee record, with fields separated by a '`:`'. Assuming that an appropriate foreign-data wrapper and foreign server have already been declared,

the following DDL statement could be employed to declare this file as a foreign table:

```
CREATE FOREIGN TABLE Personnel (
  id         INTEGER,
  last_name  VARCHAR(30),
  first_name VARCHAR(25), ...)
  SERVER myForeignServer
    OPTIONS ( Filename
       '/usr/joe/personnel.txt',
      Delimiter ':' ) ;
```

Since the foreign-data wrapper that will be used to access this file supports access to any file of this general type, each foreign table definition specifies the appropriate filename and delimiter using generic options. A user who would like to know how many people with the last name "Miller" are stored in the Personnel table could submit the following query (note that the user does not need to be aware that Personnel is a foreign table):

```
SELECT COUNT (last_name)
FROM Personnel
WHERE last_name = 'Miller';
```

The SQL-server first parses and validates this query, ensuring that it is syntactically and semantically correct and that the user has all necessary privileges. Next, it examines the `FROM` clause and discovers that it contains a reference to a foreign table. Therefore, the SQL-server establishes a connection to myForeignServer and formulates a request equivalent to the SQL statement "`SELECT last_name FROM Personnel`". The request does not include the predicate from the `WHERE` clause or the aggregate function in the `SELECT` list, but future versions of the standard will allow such requests.

The foreign-data wrapper examines the request using the foreign-data wrapper interface SQL-server routines defined by the standard to obtain the name of the referenced table, associated options, the columns to be retrieved and their types, *etc.* In this example, the wrapper returns a reply indicating that the request can be completely satisfied, as well as an execution plan. A simple implementation of the foreign-data wrapper might use Unix shell commands to extract the requested columns from the file. In this case, the execution plan would contain the relevant commands (*e.g.*, `cut -d: -f2 /usr/joe/personnel`). The SQL-server examines the reply handle and discovers that the foreign-data wrapper can completely handle the request. It incorporates the execution plan for the fragment into its overall execution plan, which must include extra processing steps to filter and aggregate the result set that will be returned by the foreign-data wrapper.

Using Datalinks

`DATALINK` is a new SQL data type that allows storing in an SQL column a reference to a file that is located in a file system external to the database system (DBS). Datalinks extend the functionality of database systems to include control over external files without the need to store their contents directly in the database. Advantages of this technology include being able to use on files the robust referential integrity, recovery, and authorization mechanism of the database management system, while avoiding the expense and application breakage caused by importing the file contents into the database (such as through LOBs). Management of files on multiple distinct file servers allows robust centralized control over distributed resources across intranets. Datalinks are very useful for any application that involves processing of information from multiple heterogeneous sources, including databases and file systems, where it is required that this information be kept consistent between the different sources.

Examples for such applications are e-commerce, customer relationship management, supply chain management, e-business applications, automotive insurance, and health insurance applications where files such as X-rays, ECG results, vehicle damage pictures, *etc*., need to be combined. It also includes CAD/CAM applications involving design documents and plans, configuration and asset management applications such as web assets (web pages, server-side programs and templates), and the management of large volumes of scientific data residing in file systems. For example, inventory data can be stored in the database while pictures of the products reside in files. To avoid a picture being accidentally removed or being replaced by another one that shows a different product, datalinks are used.

A datalink value is stored in a column of type `DATALINK`, just as a whole number would be stored in a column of type `INTEGER`. In addition to being stored as the value of a column, a datalink value can also be stored as the value of an attribute of a user-defined structured type. SQL/MED allows a variety of options to be specified for instances of the `DATALINK` type. With these options, it can be determined how strict the database controls the file. The possibilities range from no control at all (the file does not even have to exist) to full control, where removal of the datalink value from the database leads to a deletion of the physical file. Examples 2 and 3 show how a column of type `DATALINK` would be defined in a `CREATE TABLE` statement.

Example 2: The DBS does not care whether a value that will be inserted into the `DATALINK` column references a file that really exists — references

to files that do not exist will be handled appropriately by the application. All file permissions are enforced as specified at the original file system.

```
CREATE TABLE houses (
   id      INTEGER,
   name    VARCHAR(30),
   address VARCHAR(100),
   picture DATALINK
           NO LINK CONTROL );
```

Example 3: The DBS requires full control over the referenced file — that is, the DBS determines who is allowed to read the file, allows only implementation-defined updates of the file's content, and wants the file deleted when it is no longer referenced by a datalink value.

```
CREATE TABLE products (
   id      INTEGER,
   name    VARCHAR(30),
   picture DATALINK
           FILE LINK CONTROL
           INTEGRITY ALL
           READ PERMISSION DB
           WRITE PERMISSION BLOCKED
           RECOVERY YES
           ON UNLINK DELETE );
```

Even though a datalink value may "look like" a character string, it is actually an encapsulated value that contains a *logical reference* from the database to a file stored outside the database. Consequently, SQL/MED defines a function named `DLVALUE()` that generates the datalink value from a file reference, given in the form of a URL. Examples 4 and 5 illustrate the use of this function in connection with the `INSERT` statement.

Example 4:

```
INSERT INTO products
  VALUES (12, 'fender',
    DLVALUE('file://myFileServer/
         myDirectory/fender.jpg'));
```

Example 5:

```
INSERT INTO houses
  VALUES (12001,
    'villa on the hill',
    'San Jose, CA',
    DLVALUE('http://someServer
      .someCompany.com/
      file_may_not_exist/xyz.jpg'));
```

In order to "take control" of the file, the database system utilizes a component called a *datalinker*. The interaction between a database system, a datalinker, and a client application during an insert operation involving a datalink value (as shown in Example 4) is described by the following four steps:

1. The client application sends the SQL `INSERT` statement to the database system.
2. The database system contacts the datalinker to determine whether the file exists, and, if so, informs the datalinker that the file is now controlled by the database system.
3. The datalinker reports whether or not the file exists.
4. If the file could be successfully linked to the database, then the database system reports a successful execution to the client application; otherwise, it returns an error.

Figure 2 illustrates the relationships between an SQL-server, a datalinker, and a linked file.

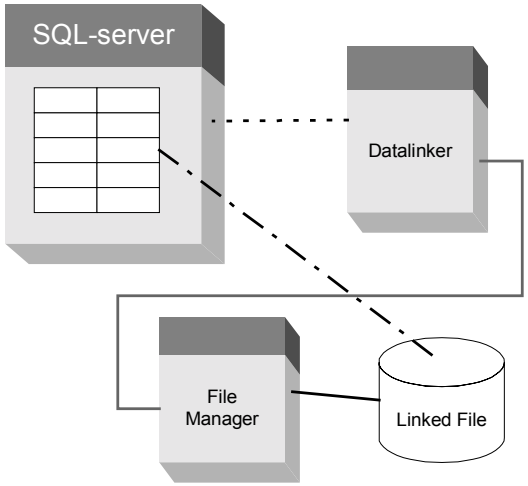

Figure 2 — Datalink-related relationships

As soon as the tables are populated, a user can execute queries against them. SQL/MED supports five functions that convert a datalink value, or parts of it, into a character string with which the user can work. For example, in order to see a picture of a product or house, the user may need to obtain a file reference for the file, then access the file directly from the file system using this file reference. The user cannot get the picture directly from the DBS, because it is not stored in the DBS! Suppose the user would like to see a picture of a car fender, of which she knows the product id. An application could execute the following statement to retrieve the necessary information:

```
SELECT DLURLPATH(picture) INTO :hv
FROM products
WHERE id = 12;
```

After successful execution of this statement, the host variable `hv` contains the proper file reference, as well as an *access token* that entitles the user to access the file. The application can now use the native API of the file server to access the file. If it does so, an `OPEN()` call from the application to the file server is *intercepted* by the datalinker. The datalinker checks the validity of the access token and, if valid, eventually grants access to the file.

One of the strengths of database systems is security. In order to extend this strength to datalinks, the datalinker has to work closely with the database system. For example, if an application attempts to open a

file with an incorrect access token (or no access token at all) and that file is referenced by a datalink value that is stored in a column declared with `READ PERMISSION DB`, then the datalinker must prohibit this access. The same holds true for all attempts to rename or delete a file (and therefore render the stored datalink value useless). If that file is referenced by a datalink value in a column declared with `INTEGRITY ALL`, then the datalinker must not allow renaming or deleting of the file.

Example 6 shows the effect of the option `ON UNLINK DELETE` of the `CREATE TABLE` statement in Example 3 when a datalink value is deleted.

Example 6: With the following `DELETE` statement, the row will be removed from the table that was inserted in Example 4.

```
DELETE FROM products WHERE id = 12;
```

Additionally, the referenced file will also be deleted, even though it existed before the row was inserted into the table.

SQL/MED's Future

The version of SQL/MED that was completed in 2000 (MED:2000) is limited in several ways. Most importantly, it provides a read-only interface to data managed by foreign servers. The next version, which will probably be published in late 2002 or early 2003, will probably add the ability to insert data into foreign tables, update such data, and delete that data.

In addition, MED:2000 required that the foreign-data wrapper support only "`SELECT * FROM foreign-table`". The next version of SQL/MED will allow an SQL-server to transmit a complete `WHERE` clause, or at least a complete predicate, to the foreign-data wrapper for evaluation. It may also be possible to instruct the foreign-data wrapper (or the foreign server) to perform joins, projections, grouping operations, and various SQL functions.

The expected datalink changes for the next version of SQL/MED will provide usability and functionality improvements. The most important change being discussed is the ability to update a linked file *and* provide file data recovery. In the current version of SQL/MED, updating a file (while the option to recover lost file data is in place) requires two steps: the first to unlink the file and the second to modify the file and re-link it. The ability to update in place will allow such updates in one step. This functionality will provide the ability to use datalinked files for such functions as library checkout and checkin and a way to back out any uncommitted file changes and restore to the previous committed version.

Summary

We have discussed SQL's new part 9, SQL/MED, which provides features that enable SQL applications to integrate non-SQL data along with data stored in an SQL-server. This facility, like much new technology, may appear complex at first glance, but we believe that it is remarkably simple for the amount of power it provides.

In fact, one of the primary goals of the SQL/MED architecture is to enable non-DBMS developers to write foreign-data wrappers for a wide variety of non-SQL data sources in as portable and rapid manner as possible. We believe that a third-party marketplace for such foreign-data wrappers, portable between various vendors' database systems, could arise within just a few years.

As soon as publication of SQL/MED has been finalized (probably late in the first quarter of 2001), copies of this part of the SQL standard can be purchased both as PDF downloads and in hardcopy from the ANSI Electronic Standards Store from the NCITS Standards Store (both cited below). In fact, all parts of the SQL standard can be acquired in the same manner. Needless to say, the PDF downloads are much less expensive than the paper versions!

References

[1] ISO/IEC 9075-9:2000, *Information technology — Database languages — SQL — Part 9: Management of External Data (SQL/MED)*, International Organization for Standardization, 2000

[2] *SQL Standardization: The Next Steps*, Andrew Eisenberg and Jim Melton, SIGMOD Record, Vol. 29, No. 1, March, 2000

[3] ISO/IEC 9075-3:1999, *Information technology — Database languages — SQL — Part 3: Call-Level Interface (SQL/CLI)*, International Organization for Standardization, 1999

[4] ISO/IEC 9075-2:1999, *Information technology — Database languages — SQL — Part 2: Foundation (SQL/Foundation)*, International Organization for Standardization, 1999

[5] Mary Tork Roth, Peter M. Schwarz, *Don't Scrap It, Wrap It! A Wrapper Architecture for Legacy Data Sources,* VLDB 1997: 266-275

Web References

[1] ANSI's Electronic Standards Store: http://webstore.ansi.org

[2] NCITS' Standards Store: http://www.cssinfo.com/ncits.html