Inform. Systems Vol. 11, No. 4, pp. 323–336, 1986
Printed in Great Britain

0306-4379/86 $3.00 + 0.00
Pergamon Journals Ltd

A DATABASE LANGUAGE FOR SETS, LISTS AND TABLES

P. PISTOR[1] and R. TRAUNMUELLER[2]

[1]IBM Deutschland GmbH, Heidelberg Scientific Center, Tiergartenstrasse 15, D-6900 Heidelberg, Federal Republic of Germany
[2]Johannes Kepler Universität, Institut für Imformatik, A-4040 Linz, Austria

(*Received* 16 *October* 1985; *in revised form* 12 *June* 1986)

1. INTRODUCTION

The classical relational model requests all data to be in first normal form (1NF) [1, 41]. While this requirement considerably simplifies the data model, it is not indispensable [3]. In the past, several proposals have been made for data models supporting unnormalized relations [4–6]. Often, they arose in environments where the simplicity of the relational model was perceived as being too restrictive. Examples are the managmeent of scientific and engineering data [7, 8], "form models" for office systems [9–11], or the integrated management of formatted and unformatted (i.e. textual) data [12].

The NF^2 model† [13] was one attempt to meet these requirements. Algebraic aspects of this and of related models have been addressed in numerous papers (e.g. [14–17, 43]). It has also been attempted to provide interfaces for those data models [10, 13, 19–22], which are inspired by the "high level" and "non-procedurality" paradigm of interfaces like SQL [23, 24] or QUEL [25]. These attempts correspond to similar ones, e.g. for network and entity relationship data models [2, 26, 27].

This paper describes a language interface for an *extended* NF^2 model, i.e. a model which covers a much broader range of data structures than the original [17] NF^2 model. While the latter one does not suffer from certain drawbacks of 1NF relations, it does not appropriately match further vital needs of new database applications (e.g. ordered items). To do so, one could allow for a few selected field level data types like vectors, possibly complemented by dedicated operators. Instead of such an asymmetric extension we prefer an orthogonal approach. For instance, tables may either be ordered (lists of tuples) or unordered (relations, i.e. sets of tuples); lists are not necessarily composed of scalars (like in Fig. 1) or of tuples, but may have elements of type "list" or "set", instead. Similarly, sets are no longer restricted to elements of type "tuple".

An SQL-like query interface for those structures has to meet quite a number of requirements. First of all, it needs to provide some equivalent of NF^2-algebra operations, like extended projection (Section 3.1) and restriction (Section 3.2), or unnesting (Section 3.6.4) and nesting (Section 3.6.5). Projection and restriction can readily be generalized to accept not only ordered tables (Section 3.4), but also any other kind of lists and sets (Section 3.5). Furthermore, this is even possible for join, nest, and unnest operations (Section 3.6). This achievement was considerably facilitated by interpreting SQL's SFW‡ construct as generic constructor for sets and lists [28], and by embedding it into a collection of appropriate expression primitives (Sections 3.3 and 3.8.1). While being the backbone of the query language proposal, we believe that the SFW construct needs to be complemented by other high-level constructs. In part, they are needed to provide capabilities outside of the SFW scope (like ordering, see Section 3.8.2), others are motivated by usability considerations (Section 3.7), but also by attempts of syntactically supporting query optimization (Sections 3.7.3, 3.8.2 and 3.8.3).

As in query support, the DML operations on NF^2 structures require more powerful facilities than those needed in classical SQL. In the latter case, only two levels are essentially affected by DML operations, the table level (insertion and deletion of tuples), or the tuple field level (update). In contrast, we need insertion and deletion operations (Sections 4.1 and 4.2), the targets of which may be any kind of lists or sets, possibly being itself part of lists, sets, or tuples. The targets of update operations (Sections 4.3 and 4.4) may be objects of any type, again either being self-contained or components of encompassing objects.

2. DATA STRUCTURES OF THE NF^2 MODEL

2.1 *An Introductory Example: Spectral Table*

The NF^2 model is concerned with two kinds of data, atomic data and composite (or non-atomic) data. The atomic data may be of type

numeric
character
Boolean

Composite data are either sets, tuples, or lists, the elements of which are either atomic or non-atomic. In case of tuples, the elements need not necessarily have the same type, whereas no element of a list or a set

†"NF^2" stands for "non first normal form".
‡Shorthand for "SELECT..FROM..WHERE".

{ SPECTRAL_TABLE }					
SUBSTANCE	{ SPECTRA }				
	SOLVENT	CONCENTRATION	<| S_GRAPH |>		
			START	STEPW	< EXTINCTION >
Indol- C-Alcaloide- D-Dichloride	N/10 KOH	5.14 10E-5	220	10	3.76 3.71 3.83 3.78 3.50 3.20 3.23 3.28 3.16 2.93 2.61
	Water	2.59 10E-5	210	10	4.60 4.12 4.09 4.17 3.92 3.61 3.49 3.67 3.61 3.29 2.90 2.50
	1N HCl	5.08 10E-5	210	10	4.87 4.36 4.24 4.29 4.12 3.90 3.87 3.94 3.88 3.73 3.54 3.38 3.12 2.80
ESTRADIOL	Methanol (80%) / NaOH pH 7.0	---	268	2	800 1250 1510 1700 1820 2000 2080 2020 1910 1860 1730 1350 660 330
	Methanol (80%) / NaOH pH 11.8	---	272	2	1080 1220 1390 1520 1650 1750 1830 1970 1990 1920 1800 1700 1670 1640 1580 1510 1400 1200 900 700 500 300
	Methanol (80%) / NaOH pH 13.1	---	270	2	750 820 910 1020 1150 1310 1490 1760 2040 2340 2500 2700 2850 2890 2880 2830 2650 2840 2020 1600 1100 800 470 330
. . . .					

Fig. 1. "SPECTRAL_TABLE": Table recording extinction spectra for chemical compounds in different solvents (data have been obtained from [44]). Extinction data are given in arbitrary units.

may differ in type from any other element of that list or set. The spectral table in Fig. 1 might serve as an illustration. This table comprises a set of rows—or tuples—representing different SUBSTANCEs and their associated SPECTRA. The SPECTRA for a given substance differ due to different SOLVENTs and substance CONCENTRATION. The spectral curves S_GRAPH are regularly sampled functions—say $f(x)$—represented by the START value x_0, the (fixed) STEPWidth $x_i - x_{i-1}$, and a list of EXTINCTION values $f(x_i)$. Thus, at the top level the SPECTRAL_TABLE can be understood as a binary relation. At the next lower level, i.e. within the tuple for a specific SUBSTANCE, the SPECTRA field again contains a relation, this time with three columns—or attributes—containing three atomic fields (START, STEPW, CONCENTRATION) and two non-atomic ones (character lists in SOLVENT, numeric vectors in EXTINCTION). Note that the restrictions on sets and lists (see above: "composite data") have been observed: all tuples within SPECTRAL_TABLE or SPECTRA are equally structured, and all lists (SUBSTANCE, SOLVENT, EXTINCTION) contain elements of just one type. This does not mean, of course, that all lists (e.g. SUBSTANCE names or spectral curves (see S_GRAPH) are of the same length.

Within this paper, examples will mainly be based on the SPECTRAL_TABLE (Fig. 1). Minor examples for other kinds of composite objects (e.g. lists of tuples, lists of lists, sets of lists, sets of sets etc.) are given as needed for the illustration of selected operations.

2.2 *Data Definition Language*

The Data Definition Language (DDL) we propose (see "Data Definition Facilities" in Appendix A1) is best illustrated by the declaration of the spectral table (given in Fig. 1):

```
(1.0) CREATE OBJECT
(1.1) SPECTRAL_TABLE{
(1.2)                  <|SUBSTANCE:CHAR,
(1.3)                    SPECTRA:{
(1.4)                              <|SOLVENT:CHAR,
(1.5)                                CONCENTRATION:REAL,
(1.6)                                S_GRAPH:<|START      :REAL,
(1.7)                                         STEPW      :REAL,
(1.8)                                         EXTINCTION:<REAL>
(1.9)                                        |>
(1.10)                             |>
(1.11)                           }
(1.12)                 |>
(1.13)               }
```

Taking into account that the pairs {...}, <...>, and <|...|> are used to indicate set, list, and tuple declarations, the previous command CREATEs the data base OBJECT SPECTRAL_TABLE as a set [cf. (1.1) and (1.13)] of tuples [cf. (1.2) and (1.12)], consisting of the character string field SUBSTANCE and the relation-valued field SPECTRA [(1.3) through (1.11)]. It is left to the reader to associate further declarations like tuple-valued [(1.6) through (1.9)] or list-valued (1.8) fields with the description given in Section 2.1.

The CREATE command allows the declaration of named sets or lists, as well as named tuples or even named atomic data base objects. Sets and variable length lists are initially empty, atomic objects are initialized by "zero" (in case of types INTEGER or REAL) or "blank" (in case of a "single character" type S_CHAR). When a tuple or a fixed length list is CREATEd, its fields are initialized according to the rules of top level sets, lists, tuples, or atoms.

As can be seen from our example, all tuple fields have been given names [e.g. SUBSTANCE in (1.2)]. As to be seen from the data definition facilities (refer to "Data Definition Facilities" in Appendix A1), naming is not mandatory. This freedom is acceptable, since we supply two addressing mechanisms for tuple fields (see Section 3.3).

Lists and sets potentially may contain any number of elements. The user may indicate a maximum number of elements, e.g.

(2) ...SPECTRAL_TABLE{..., (10000 VAR)}

or a fixed number of elements, as in

(3) EXTINCTION:<REAL, (100)>...

Contrary to lists, the declaration of sets with a *fixed* number of elements will not be allowed. This decision has been taken to keep the initialization rules simple.

3. QUERY OPERATIONS

The probably most important classes of NF^2 structures—let us call them tables for short—are sets and lists of tuples. The next three sections of this chapter are concerned with the essential operations on NF^2 tables. These operations are purposely oriented at the SFW construct of SQL-like languages (e.g. [23, 24]), but give it an interpretation such that the SFW construct, when used in a nested fashion, supports projection (Section 3.1) and restriction (Section 3.2) facilities as required for NF^2 tables. We then show how the construct can be further generalized to apply on lists and sets in general (Sections 3.4 and 3.5).

Special attention is given to join facilities as provided by properly generalized SFW expressions. They take both ordered and unordered tables, but also any other kind of lists and sets. Furthermore, joins along hierarchical paths (Section 3.6.4) are supported in a fashion which covers even more than the unnesting facilities required by the NF^2 algebra. Finally, the nest operation (Section 3.6.5) can be supported by nested SFW expressions.

In spite of its considerable syntactical resources, we think that the SFW construct should not be overloaded. For that reason, ordering, grouping (Section 3.8.2), and quantifier facilities (Section 3.8.3) are provided as dedicated constructs. However, their format has been kept quite close to the SFW format.

Masking techniques and other useful shorthand notations are treated in Sections 3.7 and 3.8.1.

3.1 *Projection*

Assume we want to retrieve the names of all SUBSTANCEs in SPECTRAL_TABLE. The query

```
(4.0) SELECT
(4.1)        ⟨|SUBSTANCE: x.SUBSTANCE|⟩
(4.2) FROM x IN SPECTRAL_TABLE
```

requests the DBMS to iterate over all tuples of the table [see (4.2)], and to construct for each tuple "x" a new tuple [denoted by tuple constructors ⟨|. .|⟩ in (4.1)] by just using the SUBSTANCE field value. In our example, the field of the resulting tuple is given the same field identifier as before.

We could equally well have chosen the SPECTRA column for projection; by additionally renaming the column, we end up with the expression.

```
(5.0) SELECT⟨|SPECTRUM:x.SPECTRA|⟩
(5.1) FROM x in SPECTRAL_TABLE
```

Except for the mandatory use of the tuple identifiers and tuple constructors, projection expressions like (4) and (5) are very similar to those used in languages like SQL [24]. The situation is different, however, when we want to retrieve, for example, for any SUBSTANCE the names of the SOLVENTs and the CONCENTRATIONs. In this case, a projection has to be performed *inside* the SPECTRA values

x.SPECTRA

of every tuple x in SPECTRAL_TABLE:

```
(6.0) SELECT
(6.1)        ⟨| x.SUBSTANCE,
(6.2)         (SELECT
(6.3)                ⟨| y.SOLVENT,
(6.4)                   y.CONCENTRATION|⟩
(6.5)          FROM y IN x.SPECTRA)
(6.6)        |⟩
(6.7) FROM x IN SPECTRAL_TABLE
```

In this nested projection we have intentionally omitted field names in the tuple expressions [see (6.1), (6.2), (6.3) and (6.4)]. This point will be discussed in Section 3.3.

3.2. *Restriction*

Let us suppose that we want to retrieve those tuples x of the SPECTRAL_TABLE which contain at least one spectrum for the SOLVENT 'C_2H_5OH'. The formulas (7) and (8) show two semantically equivalent queries serving that purpose:

```
(7.0) SELECT x
(7.1) FROM x IN SPECTRAL_TABLE
(7.2) WHERE {}¬= 
(7.3)        (SELECT xx.SOLVENT
(7.4)          FROM xx IN x.SPECTRA
(7.5)         WHERE xx.SOLVENT = 'C2H5OH')
```

Note: The tuple constructors have intentionally been omitted in (7.0) and (7.3) (see also Section 3.3).

```
(8.0) SELECT x
(8.1) FROM x IN SPECTRAL_TABLE
(8.2) WHERE (EXISTS xx IN x.SPECTRA:
(8.3)                xx.SOLVENT = 'C2H5OH')
```

While equations (7) and (8) are quite close to SQL-like queries on flat tables, this is no longer true if we modify our query such that only SPECTRA entries for the SOLVENT "alcohol" are to be returned:

```
(9.0) SELECT
(9.1)    ⟨| x.SUBSTANCE,
(9.2)      (SELECT xx
(9.3)       FROM   xx IN x.SPECTRA
(9.4)       WHERE xx.SOLVENT = 'C2H5OH')|⟩
(9.5) FROM x IN SPECTRAL_TABLE
(9.6) WHERE (EXISTS z IN x.SPECTRA
(9.7)                z.SOLVENT = 'C2H5OH')
```

In this query, we nest restrictions in a similar way as we have nested projections in query (6). Of course, this kind of nesting would not make sense for flat tables. [By the way: If we would not bother about possibly empty SPECTRA fields in the query result, we could omit the WHERE-clause (9.6–7) in the previous query.]

3.3 *Tuple Expressions*

In the previous section, we have encountered operations like

constructing tuples [e.g. example (6)],
accessing tuple field values [e.g. (6.1)],
naming tuple fields [e.g. (4.1)].

In this section, tuple operations will be discussed in some more detail.

Let z denote the tuple

(10) ⟨|'H_2O', 10|⟩ (note the different types!)

The value of the first field is 'H_2O', the value of the second field is 10. That is, in accordance with other SQL-like languages, field order in tuples is relevant. Therefore, we may access field values by the ordinal numbers, e.g.

$z.1$ (yielding 'H_2O'), or
$z.2$ (yielding 10).

According to the DDL syntax (see 'Data Definition Facilities" in Appendix A1), tuple fields may option-

ally be named. Accordingly we might *construct* (see "Tuple Related Expressions" in Appendix A3.3) a tuple, say z', as

(10') ⟨|SOLVENT: 'H_2O', CONCENTRATION: 10|⟩

We say that the fields of tuple (10) are *anonymous*; in contrast, the fields of tuple (10') are explicitly named.

We may access e.g. the second field in tuple z' in two alternative fashions, both resulting in the same value:

(11.1) $z'.2$
(11.2) z'.CONCENTRATION

The important fact in expression (11) is, that access to a tuple field gives access to the field *value*. The field *name* is lost. Therefore, the fields of the result tuples constructed in (6.1) (first field) and (6.2) through (6.6) (second field) are anonymous, as well as the fields of the lower level tuples [see (6.3) and (6.4)].
Query expression (5) is an example of a query in which the resulting tuples are constructed such that their field names (here: actually one field at the top level) do *not* remain anonymous:

(5) SELECT⟨|SPECTRUM:x.SPECTRA|⟩...

If we compare query (5) with the input SPECTRAL_TABLE, the question arises whether the lower level tuples of the result will be composed of anonymous tuples or not. The answer is "no", since the repeating groups, i.e. the x.SPECTRA field values, are taken as a whole. For the same reason, the result of query (7) does not contain anonymous tuples, since the resulting table is constructed from *complete* tuples x of the input SPECTRAL_TABLE.

Conventions on the treatment of tuple field names are also required in the context of certain set/list operations (see e.g. UNION, CAT in Section 3.8) or in case of insertion and update operations (see Section 4). Details need not be discussed here.

3.4. *Queries Involving Lists*

In the examples presented so far, the "SELECT...FROM...WHERE" construct has been applied on sets of tuples, and these operations in turn resulted in sets of tuples. On the other hand, we could have created, instead of SPECTRAL_TABLE [see (1)], a list-type equivalent, say SPECTRAL_TABLE_L, as

```
(12.0) CREATE OBJECT
(12.1) SPECTRAL_TABLE_L
(12.2)   ⟨
(12.3)     ⟨|SUBSTANCE:CHAR,
(12.4)       SPECTRA   :⟨
(12.5)                   ⟨|SOLVENT          :CHAR,
(12.6)                     CONCENTRATION:REAL,
(12.7)                     S_GRAPH         :⟨|START      :REAL,
(12.8)                                       STEPW      :REAL,
(12.9)                                       EXTINCTION:⟨REAL⟩
(12.10)                                     |⟩
(12.11)                  |⟩
(12.12)                ⟩
(12.13)    |⟩
(12.14)  ⟩
```

The examples seen so far would also apply on this new object. However, the operations would result in lists of tuples rather than sets of tuples. Different from sets, the list elements would be accessed in list order, and the order of the input table would therefore be preserved in the result.

Other than sets, lists allow duplicates. Therefore, duplicates resulting from query operations (e.g. projection on a list of tuples) will *not* be suppressed. This may be important e.g. when queries are supposed to provide statistical information.

The question might arise whether it is possible to join lists of tuples, or even lists of tuples with sets of tuples. This question will be addressed in Sectins 3.6.2 and 3.6.3.

In the remainder of the current section, we will see that we can also deal with other types of lists than just lists of tuples. For that end, let

(13) $V = \langle 1, 2, 5, 4, 8 \rangle$

If we want to retrieve from V all elements which are succeeded by a greater one, we could do so by

```
(14.0) SELECT
(14.1)          V[i]
(14.2) FROM     i IN INDL(V)
(14.3) WHERE    i < LEN(V) AND
(14.4)          V[i] < V[i + 1]
```

Here, "$V[i]$" addresses the ith element of "V". "INDL" is a built-in function which takes a list and returns an index vector 1, 2, ... The elements "i" of this vector are accessed one after the other, and a check is made whether there exists a successor element "$i + 1$", such that "$i + 1$" does not exceed the length of "V". If so, "$V[i]$" is compared with "$V[i + 1]$". The acceptable values "$V[i]$" are SELECTed for the resulting list.

3.5 *Queries Involving General Sets*

For the sake of completeness we would like to demonstrate that the SFW-construct is able to deal also with sets of "non-tuples". Let

(15) $S = \{1, 2, 7, 8, 10, 13\}$

{TAB_S1}		
C1	C2	C3
1	5	2
2	8	3
3	9	2

```
CREATE OBJECT
    TAB_S1 { <| C1: INTEGER,
                C2: INTEGER,
                C3: INTEGER |> }
```

Fig. 2. "TAB_S1": Table with 3 integer type attributes C1, C2, C3. The corresponding CREATE command is also given.

{RES_Q17}			
C1	C2	C3	C4
1	5	2	2
1	5	2	3
2	8	3	3

Fig. 3. "RES_Q17": Join result of query (17); the table represents a set of tuples.

To retrieve from "S" all elements greater than 9, we write

```
(16.0) SELECT
(16.1)        x
(16.2) FROM   x IN S
(16.3) WHERE  x > 9
```

Since "S" is composed of integers, the free variable x stands for 1, 2, 7, ..., or 13, in turn, when evaluating the query. Thus, according to our rules, the result of the query is again a set of integers.

3.6 *Joins*

3.6.1 *Classical joins*

Let us consider a simple ternary table TAB_S1 as sketched in Fig. 2. Using TAB_S1, let us generate a new table (see Fig. 3) in which we combine each tuple x of TAB_S1 with all the tuples of that table having a greater C1-value than the tuple x. This result is achieved by the join operation.

```
(17.0) SELECT
(17.1)        <|C1:x.C1,
(17.2)          C2:x.C2,
(17.3)          C3:x.C3,
(17.4)          C4:y.C1|>
(17.5) FROM x IN TAB_S1, y IN TAB_S1
(17.6) WHERE x.C1 < y.C1
```

3.6.2 *Joining ordered tables*

Rather than having a *set* TAB_S1 of tuples, let us assume a *list* TAB_L1 of tuples (see Fig. 4). Obviously, the previously discussed join operation can be generalized such that it applies not only on *sets* of tuples, but also on *lists* of tuples. Starting with the first tuple, we try to combine it—in sequence—with the first, second and third tuple of "TAB_L1". Then we do the same with the remaining ones to finally obtain Fig. 5.

The corresponding query statement is completely analogous to the set case (17):

```
(18.0) SELECT
(18.1)        <|A1:x.A1,
(18.2)          A2:x.A2,
(18.3)          A3:x.A3,
(18.4)          A4:y.A1|>
(18.5) FROM   x IN TAB_L1, y IN TAB_L1
(18.6) WHERE  x.A1 < y.A1
```

As can be seen from the description given above, the evaluation is different for (17) and (18). With sets (here: TAB_S1) there is no predefined order in which the set elements are accessed. With lists, the order is important in a twofold way: First, the elements of each input list are accessed in list order. Second, the FROM-clause is logically equivalent to a sequence of nested loops: the left-most list being addressed in the FROM-clause defines the outermost loop, while the right-most list defines the innermost loop (see Figs 4 and 5).

3.6.3 *Joins involving both ordered and unordered tables*

The rules given so far do not cover the join between lists and tables, as in

```
(19.0) SELECT
(19.1)        <|A1:x.A1,
(19.2)          A2:x.A2,
(19.3)          A3:x.A3,
(19.4)          A4:y.C1|>
(19.5) FROM   x IN TAB_L1, y IN TAB_S1
(19.6) WHERE  x.A1 < y.C1
```

Several ways are conceivable to fix the semantics of this expression. Our proposal is captured in the following rules:

<TAB_L1>		
A1	A2	A3
1	5	2
2	8	3
3	9	2

```
CREATE OBJECT
    TAB_L1 < <| A1: INTEGER,
                A2: INTEGER,
                A3: INTEGER |> >
```

Fig. 4. "TAB_L1": List of tuples (ordered table), and corresponding CREATE command.

<RES_Q18>			
A1	A2	A3	A4
1	5	2	2
1	5	2	3
2	8	3	3

Fig. 5. "RES_Q18": Result of join operation (18). Different from Fig. 3, this table represents a *list* of tuples. Therefore, order is important.

1. If at least one list is involved in the FROM-clause, the result is a list.
2. The *order* of the result elements is only partly determined, if not all of the operands in the FROM-clause are lists.
3. If at least one list is involved in the FROM-clause, then the FROM-clause *must* be evaluated from left to right, such that for any two operands (e.g. TAB_L1, TAB_S1) the iteration is slower for the left one ("outer loop"), and faster for the right one ("inner loop").
4. List elements are accessed in list order, set elements are accessed in any order.

3.6.4 *Unnesting by joins*

Let us return to our SPECTRAL_TABLE (Fig. 1). How could we "flatten" this table in such a way that the SUBSTANCE name is repeated for any of its SPECTRA? Here is an answer:

```
(20.0) SELECT⟨|SUBSTANCE:x.SUBSTANCE,
(20.1)          SPECTRUM:y                |⟩
(20.2) FROM  x IN SPECTRAL_TABLE,
(20.3)       y IN x.SPECTRA
```

In this kind of join, each element of the Cartesian product in (20.2–3) is *conceptually* formed by a complete top level tuple and one of its SPECTRA tuples. This kind of join will sometimes be referred to as "implicit join" (see also [29]).

A note is in place here on the Cartesian product given by (20.2–3). Each single tuple x of SPECTRAL_TABLE is—conceptually—replicated as often as we have tuples in the SPECTRA field. Therefore, the cardinality of the Cartesian product is given by the total number of SPECTRA tuples. Consequently, a tuple must be discarded if the SPECTRA field is empty. This is in accordance with the fact that a Cartesian product of two sets is empty if at least one of the sets is empty.

Instead of using the implicit join feature, the result of expression (20) could also be achieved by using the DUNION ("distributed union" operation, see Section 3.8.1):

```
(20.0') DUNION(
(20.1')   SELECT(
(20.2')          SELECT⟨|SUBSTANCE:x.SUBSTANCE,
(20.3')                 SPECTRUM:y|⟩
(20.4')           FROM y IN x.SPECTRA )
(20.5')   FROM x IN SPECTRAL_TABLE )
```

It is felt, however, that this approach is less elegant and also harder to understand.

3.6.5 *Establishing hierarchies*

In the previous example, the join served as a means to flatten a hierarchy. The reverse effect can be achieved as well. Let the flat table "FT" denote the result of (20). From the table "FT", the contents of SPECTRAL_TABLE are recovered as follows:

```
(21.0) SELECT
(21.1)        ⟨|SUBSTANCE: x.SUBSTANCE,
(21.2)         SPECTRA: (SELECT y.SPECTRUM
(21.3)                   FROM y IN FT
(21.4)                   WHERE x.SUBSTANCE = y.SUBSTANCE)
(21.5)        |⟩
(21.6) FROM x IN FT
```

Obviously, expression (21) is not a typical join expression, but some new kind of nested query. However, in analogy to its inverses (20) and (20'), it seems quite right to discuss this nesting feature along with join operations.

3.6.6 *Joins of general sets or lists*

Up to now, we have dealt with joins of tables, only. However, joins of general sets or lists make sense as well. This can be demonstrated by rewriting expression (14) in form of a join between vectors:

```
(22.0) SELECT
(22.1)        V[i]
(22.2) FROM   i IN INDL(V), j IN INDL(V)
(22.3) WHERE  i + 1 = j
(22.4)        AND
(22.5)        V[i] < V[j]
```

3.7. *Ease-of-Use Constructs*

3.7.1 *Handling of context conditions: Masking in lists*

By context conditions we denote conditions where the relative position of certain list elements to each other is important. We have already been confronted with context conditions in Section 3.4 (example 14) and in Section 3.6.6 (example 22).

An important class of context conditions is encountered with predicates on text fields. For instance, let us assume that we look for all tuples in SPECTRAL_TABLE, where the SUBSTANCE field somewhere contains the character string 'SALICILO'.

Rather than writing

```
(23.0) SELECT x
(23.1) FROM x IN SPECTRAL_TABLE
(23.2) WHERE EXISTS (i IN INDL (x.SUBSTANCE):
(23.3)                 i ≤ 1 + LEN(x.SUBSTANCE) – 8
(23.4)                                AND
(23.5)                 SUBLIST (i, 8, x.SUBSTANCE) = 'SALICILO')
```

("INDL", "SUBLIST", "LEN" are explained in Section 3.8.1), we simply write [24]

```
(24.0) SELECT x
(24.1) FROM   x IN SPECTRAL_TABLE
(24.2) WHERE x.SUBSTANCE LIKE '*SALICILO*'
```

Here, "*" represents 0, 1, or more characters, or in more general terms, one or more arbitrary list elements.

In addition to the masking (or don't care) symbol "*", we also use the masking symbol "_". The latter represents just one arbitrary list element.

Obviously, the masking technique of the LIKE feature can be used in a more general way than indicated in [24]. Assume "x" to be a list of author names. If we want to ask, whether this list contains two subsequent names, the first of which is 'SCHULTZ', whereas the second one must end by 'MUELLER', then this condition could be specified as

(25) x LIKE⟨*, 'SCHULTZ', '*MUELLER',*⟩

This generalized LIKE expression features multilevel masking in nested lists.

3.7.2 *Masking in sets and tuples*

The masking technique can be further generalized to simplify conditions on sets and tuples:

Then,

(26) S LIKE {1, _}

tests, whether "S" contains exactly two elements, one of which is "1". In

(27) S LIKE {1, _, *}

S must contain at least 2 elements, one of which is "1".

With tuples, "_" denotes one field, "*" may be used to denote trailing fields which need not be checked. Thus,

(28) T LIKE ⟨|_, _, 5|⟩

tests, whether a tuple "T" of length 3 contains "4" as its last field value. Similarly,

(29) T LIKE ⟨|'ABC', *|⟩

tests "T" on 'ABC' in its first field, while all subsequent fields are irrelevant for the test.

3.7.3 *Common sub-expressions*

If NF^2 structures have a deep nesting structure, queries may contain a considerable number of subqueries. To avoid a negative impact on readability, suitable subexpressions should be factored out. This can be done by the USE construct, which is offered in two variants.

In the first variant, USE provides the facility to define auxiliary values. For example, instead of

```
(30.0) SELECT y
(30.1) FROM   y IN
(30.2)            (SELECT⟨|SUBSTANCE: x.SUBSTANCE,
(30.3)                     SPECTRA:    (SELECT z
(30.4)                                  FROM z IN x.SPECTRA)
(30.5)                     |⟩
(30.6)            FROM x IN SPECTRAL_TABLE)
(30.7) WHERE y.SUBSTANCE LIKE 'AMINO*'
```

we could write

```
(31.0) USE
(31.1) AUXTAB = SELECT⟨|SUBSTANCE: x.SUBSTANCE,
(31.2)                  SPECTRA:    (SELECT z
(31.3)                               FROM z IN x.SPECTRA)
(31.4)                  |⟩
(31.5)          FROM x IN SPECTRAL_TABLE
(31.6) IN
(31.7) SELECT y
(31.8) FROM   y IN AUXTAB
(31.9) WHERE y.SUBSTANCE LIKE 'AMINO*'
```

For example, let "S" denote a set of integers.

Obviously, this feature is especially helpful in case of common subexpressions.

The second variant of USE allows one to specify auxiliary functions, e.g. parametrized queries. Assume for instance that we ask for all substances having at least one spectrum with water as solvent. For the qualifying substances, we want to see only the spectra with water as solvent. Using the features discussed so far, we could specify:

```
(32.0) SELECT
(32.1)   ⟨|x.SUBSTANCE,
(32.2)    (SELECT z
(32.3)     FROM   z IN x.SPECTRA
(32.4)     WHERE  z.SOLVENT = 'H2O')|⟩
(32.5) FROM x IN SPECTRAL_TABLE
(32.6) WHERE {}¬ = (SELECT y
(32.7)                FROM y in x.SPECTRA
(32.8)               WHERE y.SOLVENT = 'H2O')
```

A more elegant way would be

```
(33.0) USE
(33.1) F(solv, y) = SELECT z
(33.2)               FROM z IN y.SPECTRA
(33.3)              WHERE z. SOLVENT = solv
(33.4) IN
(33.5) SELECT
(33.6)        ⟨|x.SUBSTANCE,
(33.7)          F('H2O', x)|⟩
(33.8) FROM     x IN SPECTRAL_TABLE
(33.9) WHERE {}¬ = F('H2O', x)
```

3.8 *Other Query Operations*

3.8.1 *Operation primitives*

This subsection summarizes primitive operations; most of them have already been presented in the preceding text. Where appropriate, the meaning of the operations is given in terms of illustrative equalities. Further detail about their semantics as well as additional operations may be found in [28, 30, 35].

3.8.1.1 *Operations on tuples.*

(34.1) ⟨| SOLVENT:'H_2O', CONCENTRATION:10 |⟩ (tuple construction)

Expressions (34.2) feature access to tuple field values. The expressions given here all return "10" as result:

(34.2) ⟨| 'H_2O', 10|⟩.2 = 10
⟨| SOLVENT:'H_2O', CONCENTRATION:10 |⟩.CONCENTRATION = 10
⟨| SOLVENT:'H_2O', CONCENTRATION:10 |⟩.2 = 10

(34.3) STRIP (⟨| 1 |⟩) = 1 = STRIP (⟨| THE_ONLY_FIELD: 1 |⟩)
(An alternative technique to access the value of a one-field tuple)

(34.4) ⟨| F1: 1, F2:2 |⟩ ‖ ⟨| F3: 3 |⟩ =
⟨| F1: 1, F2:2, F3: 3 |⟩
(Tuple concatenation)

3.8.1.2 Operations on lists.

(35.1) ⟨1, 5, 8, 3, 7⟩ (list construction)
(35.2) ⟨1, 5, 2⟩ [2] = 5
(access to list elements by subscripting)
(35.3) INDL (⟨1, 5, 2⟩) = ⟨1, 2, 3⟩ (index list generation)
(35.4) ⟨1, 2⟩ ‖ ⟨2, 3⟩ = ⟨1, 2, 2, 3⟩ (list concatenation)
(35.5) DCAT (⟪1, 2⟩, ⟨2, 3⟫) = ⟨1, 2, 2, 3⟩
(distributed concatenation)
(35.6) SUBLIST (2, 3,⟨1, 2, 3, 4, 5, 6⟩) = ⟨2, 3, 4⟩
(extraction of sublists)
(35.7) STRIP (⟨1⟩) = 1
(an alternative way of accessing the element of a one-element list)
(35.8) ELEMS (⟨1, 1, 2, 2, 4, 0⟩) = {1, 2, 4, 0}
(35.9) LEN (⟨1, 2, 3, 3⟩) = 4 (returns the length of a list)

3.8.1.3 *Operations on sets*

(36.1) {1, 2, 5, 8} (set construction)
(36.2) {1, 2} UNION {2, 3, 4} = {1, 2, 3, 4} (set union)
(36.3) {1, 2} INTERSECT {2, 3} = {2} (set intersection)
(36.4) {2, 3, 4} MINUS {2} = {3, 4} (set difference)
(36.5) ELEMS (MLIST ({1, 2, 3, 4})) = {1, 2, 3, 4}
(MLIST takes a set and returns a list;the system is allowed to return the input elements in any order)
(36.6) DUNION ({{1, 2}, {2, 3}, {3, 4}}) = {1, 2, 3, 4}
(distributed union [40])
(36.7) STRIP ({1}) = 1
(returns the element of a single-element set)
(36.8) CARD ({1, 2, 3}) = 3
(returns number of set elements)

3.8.2 *Ordering, grouping*

The ORDER operation is restricted to lists. For instance,

(37) ORDER x IN ⟨1, 3, 2, 5, 0⟩ BY x ASC

returns

⟨0, 1, 2, 3, 5⟩

Ordering may be performed on any value derivable from the list elements which is suitable to define order, e.g.

(38) ORDER x IN SPECTRAL_TABLE_L
BY CARD (x.SPECTRA) DESC

or

(39) ORDER x IN ⟪1, 2⟩, ⟨0, 1⟩, ⟨5, 3⟫
BY SUM (x) ASC

In the last example, the sublist sums (1, 3, and 8, respectively) and are used to produce

⟪0, 1⟩, ⟨1, 2⟩⟨5, 3⟫

An operation with similar syntactical structure is the grouping operation. It takes a set or list and partitions it into subsets and sublists, respectively. For instance,

(40) GROUP x IN {⟨|1, 2|⟩, ⟨|2, 5|⟩, ⟨|1, 1|⟩} BY x.1

returns

(41) {{⟨|1, 2|⟩, ⟨|1, 1|⟩}, {⟨|2, 5|⟩}}

When the GROUP operation is applied on a set, the result may be undone by DUNION. With lists, however, such a pair of operations does not exist. If

GROUP is applied on a list, a subsequent DCAT operation may return any permutation of the original list.

The effect of GROUPing can also be obtained by a SELECT expression: Instead of (40), we have (using $S = \{\langle|1, 2|\rangle, \langle|2, 5|\rangle, \langle|1, 1|\rangle\}$)

```
(42.0) SELECT (SELECT y
(42.1)          FROM y IN S
(42.2)          WHERE x.1 = y.1)
(42.3) FROM x IN S
```

One should note the similarity of (42) with the nesting operation (21). While being *semantically* equivalent to (40), it is felt, however, that expression (40) is more transparent than expression (42). It might also be simpler to optimize the evaluation of GROUP expressions rather than their SELECT equivalents.

3.8.3 *Quantified Boolean expressions*

In our query language, Boolean expressions mainly serve as filtering or restriction conditions [see, e.g. (7) or (8)]. Boolean expressions can be roughly subdivided into three categories,

(a) comparison expressions,
(b) prefixed and infix Boolean expressions,
(c) quantified Boolean expressions.

Examples of comparison expressions are

"ordinary" comparison operations (e.g. 17 = 5, B = C),
set comparison operations (e.g. A SUBSET_OF B, 1 ELEMENT_OF A),
LIKE comparison operations (see Section 3.2.7).

Category (b) involves negations and the Boolean operators "AND" and "OR" [e.g. (22)].

Examples for quantified Boolean expressions have already been given in (8), (9), and (23). Here, we shall briefly describe further types of quantified expressions.

While

```
(43.1) EXISTS xx in x.SPECTRA:
(43.2)          xx.SOLVENT = 'C2H5OH'
```

is true if at least one item '*xx*' meets condition (43.2), the following expression

```
(44.1) FOR_ALL xx in x.SPECTRA:
(44.2)          xx.SOLVENT = 'C2H5OH'
```

means that (44.2) must be met by *all* items '*xx*' of '*x*.SPECTRA'.

Between these two extremes, we offer the possibility of stating that exactly N [see (45), e.g. $N = 5$], at least N [see (46)], or at most N [see (47)] items must meet a certain condition:

```
(45.1) EXIST (EXACTLY 5) xx IN x.SPECTRA:
(45.2)                   xx.SOLVENT = 'C2H5OH'

(46.1) EXIST (AT_LEAST 5) xx IN x.SPECTRA:
(46.2)                    xx.SOLVENT = 'C2H5OH'

(47.1) EXIST (AT_MOST 5) xx IN x.SPECTRA:
(47.2)                   xx.SOLVENT = 'C2H5OH'
```

4. DATA MANIPULATION FACILITIES

Data manipulation facilities (DMF) operations comprise

deletion operations,
insertion operations,
update operations.

DMF operations can further be subdivided into basic DMF operations and compound DMF operations. The distinction is most readily explained with delete operations.

4.1 *Delete Operations*

Delete operations are intended to remove set or list elements.

Assume some list "L". If we want to erase the second element—assuming of course the length of L is not smaller than 2—we can achieve this by

```
(48) DELETE L[2]
```

The operation is an example of a *basic* deletion operation; it mainly consists of two parts, the keyword (here: DELETE) and the denotation of the object to be deleted (here: the second element of "L"). This mechanism is not able, however, to address set or list elements by their contents. For example, we cannot remove all negative items. For that end, a compound operation, the "FOR_EACH" construct, is provided:

```
(49.0) FOR_EACH
(49.1) x IN L WHERE x < 0
(49.2) DO
(49.3)     DELETE x
(49.4) END
```

It enables one to associatively address [see (49.1)] the components of a data base object which are to be operated upon. The operation itself is specified in the DO-END part [see (49.2–(49.4)]. In the DO-END part, we may encounter primitive DMF operations, as in (49.3), or again a FOR_EACH operation. The latter possibility enables one to delete lower level set or list elements, as in

```
(50.0) FOR_EACH
(50.1) item IN SPECTRAL_TABLE
(50.2) DO
(50.3)     FOR_EACH
(50.4)     spectrum IN item. SPECTRA
(50.5)     WHERE spectrum. SOLVENT = 'H2O'
(50.6)     DO
(50.7)         DELETE spectrum
(50.8)     END
(50.9) END
```

In (50), no top level tuple is removed from the SPECTRAL_TABLE. Instead, only tuples of the SPECTRA field are deleted, provided they are associated with the SOLVENT "water".

<RES_Q18>			
A1	A2	A3	A4
1	5	2	2
1	5	2	3
2	8	3	3

Fig. 5. "RES_Q18": Result of join operation (18). Different from Fig. 3, this table represents a *list* of tuples. Therefore, order is important.

1. If at least one list is involved in the FROM-clause, the result is a list.
2. The *order* of the result elements is only partly determined, if not all of the operands in the FROM-clause are lists.
3. If at least one list is involved in the FROM-clause, then the FROM-clause *must* be evaluated from left to right, such that for any two operands (e.g. TAB_L1, TAB_S1) the iteration is slower for the left one ("outer loop"), and faster for the right one ("inner loop").
4. List elements are accessed in list order, set elements are accessed in any order.

3.6.4 *Unnesting by joins*

Let us return to our SPECTRAL_TABLE (Fig. 1). How could we "flatten" this table in such a way that the SUBSTANCE name is repeated for any of its SPECTRA? Here is an answer:

```
(20.0) SELECT <|SUBSTANCE:x.SUBSTANCE,
(20.1)          SPECTRUM:y                |>
(20.2) FROM  x IN SPECTRAL_TABLE,
(20.3)       y IN x.SPECTRA
```

In this kind of join, each element of the Cartesian product in (20.2–3) is *conceptually* formed by a complete top level tuple and one of its SPECTRA tuples. This kind of join will sometimes be referred to as "implicit join" (see also [29]).

A note is in place here on the Cartesian product given by (20.2–3). Each single tuple x of SPECTRAL_TABLE is—conceptually—replicated as often as we have tuples in the SPECTRA field. Therefore, the cardinality of the Cartesian product is given by the total number of SPECTRA tuples. Consequently, a tuple must be discarded if the SPECTRA field is empty. This is in accordance with the fact that a Cartesian product of two sets is empty if at least one of the sets is empty.

Instead of using the implicit join feature, the result of expression (20) could also be achieved by using the DUNION ("distributed union" operation, see Section 3.8.1):

```
(20.0') DUNION(
(20.1')   SELECT(
(20.2')           SELECT<|SUBSTANCE:x.SUBSTANCE,
(20.3')                  SPECTRUM:y|>
(20.4')           FROM y IN x.SPECTRA )
(20.5')   FROM x IN SPECTRAL_TABLE )
```

It is felt, however, that this approach is less elegant and also harder to understand.

3.6.5 *Establishing hierarchies*

In the previous example, the join served as a means to flatten a hierarchy. The reverse effect can be achieved as well. Let the flat table "FT" denote the result of (20). From the table "FT", the contents of SPECTRAL_TABLE are recovered as follows:

```
(21.0) SELECT
(21.1)         <|SUBSTANCE: x.SUBSTANCE,
(21.2)           SPECTRA: (SELECT y.SPECTRUM
(21.3)                     FROM y IN FT
(21.4)                     WHERE x.SUBSTANCE = y.SUBSTANCE)
(21.5)         |>
(21.6) FROM x IN FT
```

Obviously, expression (21) is not a typical join expression, but some new kind of nested query. However, in analogy to its inverses (20) and (20'), it seems quite right to discuss this nesting feature along with join operations.

3.6.6 *Joins of general sets or lists*

Up to now, we have dealt with joins of tables, only. However, joins of general sets or lists make sense as well. This can be demonstrated by rewriting expression (14) in form of a join between vectors:

```
(22.0) SELECT
(22.1)        V[i]
(22.2) FROM   i IN INDL(V), j IN INDL(V)
(22.3) WHERE  i + 1 = j
(22.4)        AND
(22.5)        V[i] < V[j]
```

3.7. *Ease-of-Use Constructs*

3.7.1 *Handling of context conditions: Masking in lists*

By context conditions we denote conditions where the relative position of certain list elements to each other is important. We have already been confronted with context conditions in Section 3.4 (example 14) and in Section 3.6.6 (example 22).

An important class of context conditions is encountered with predicates on text fields. For instance, let us assume that we look for all tuples in SPECTRAL_TABLE, where the SUBSTANCE field somewhere contains the character string 'SALICILO'.

Rather than writing

```
(23.0) SELECT x
(23.1) FROM x IN SPECTRAL_TABLE
(23.2) WHERE EXISTS (i IN INDL (x.SUBSTANCE):
(23.3)                i ≤ 1 + LEN(x.SUBSTANCE) − 8
(23.4)                          AND
(23.5)                SUBLIST (i, 8, x.SUBSTANCE) = 'SALICILO')
```

("INDL", "SUBLIST", "LEN" are explained in Section 3.8.1), we simply write [24]

```
(24.0) SELECT x
(24.1) FROM   x IN SPECTRAL_TABLE
(24.2) WHERE x.SUBSTANCE LIKE '*SALICILO*'
```

Here, "*" represents 0, 1, or more characters, or in more general terms, one or more arbitrary list elements.

In addition to the masking (or don't care) symbol "*", we also use the masking symbol "_". The latter represents just one arbitrary list element.

Obviously, the masking technique of the LIKE feature can be used in a more general way than indicated in [24]. Assume "x" to be a list of author names. If we want to ask, whether this list contains two subsequent names, the first of which is 'SCHULTZ', whereas the second one must end by 'MUELLER', then this condition could be specified as

(25) x LIKE⟨*, 'SCHULTZ', '*MUELLER',*⟩

This generalized LIKE expression features multilevel masking in nested lists.

3.7.2 *Masking in sets and tuples*

The masking technique can be further generalized to simplify conditions on sets and tuples:

Then,

(26) S LIKE {1, _}

tests, whether "S" contains exactly two elements, one of which is "1". In

(27) S LIKE {1, _, *}

S must contain at least 2 elements, one of which is "1".

With tuples, "_" denotes one field, "*" may be used to denote trailing fields which need not be checked. Thus,

(28) T LIKE ⟨|_, _, 5|⟩

tests, whether a tuple "T" of length 3 contains "4" as its last field value. Similarly,

(29) T LIKE ⟨|'ABC', *|⟩

tests "T" on 'ABC' in its first field, while all subsequent fields are irrelevant for the test.

3.7.3 *Common sub-expressions*

If NF^2 structures have a deep nesting structure, queries may contain a considerable number of subqueries. To avoid a negative impact on readability, suitable subexpressions should be factored out. This can be done by the USE construct, which is offered in two variants.

In the first variant, USE provides the facility to define auxiliary values. For example, instead of

```
(30.0) SELECT y
(30.1) FROM   y IN
(30.2)            (SELECT⟨|SUBSTANCE: x.SUBSTANCE,
(30.3)                     SPECTRA:   (SELECT z
(30.4)                                 FROM z IN x.SPECTRA)
(30.5)                     |⟩
(30.6)            FROM x IN SPECTRAL_TABLE)
(30.7) WHERE y.SUBSTANCE LIKE 'AMINO*'
```

we could write

```
(31.0) USE
(31.1) AUXTAB = SELECT⟨|SUBSTANCE: x.SUBSTANCE,
(31.2)                  SPECTRA:   (SELECT z
(31.3)                             FROM z IN x.SPECTRA)
(31.4)                  |⟩
(31.5)          FROM x IN SPECTRAL_TABLE
(31.6) IN
(31.7) SELECT y
(31.8) FROM   y IN AUXTAB
(31.9) WHERE y.SUBSTANCE LIKE 'AMINO*'
```

For example, let "S" denote a set of integers. Obviously, this feature is especially helpful in case of common subexpressions.

4.2 *Insertion Operations*

We distinguish two types of basic insertion operations,

insertion into sets (INSERT),
insertion into lists (EXTEND).

With list insertions, the user is requested to specify not only the values of the elements to be inserted [see 51.1) below], but also the position where these elements shall go (51.2):

```
(51.0) EXTEND L
(51.1) WITH ⟨5, 6⟩
(51.2) AFTER 3
```

With sets, position information would be meaningless. Adding the values 5 and 6 to a set S of integers is simply achieved by

```
(52.1) INSERT {5, 6}
(52.2) INTO S
```

Again, insertion requires a FOR_EACH variant, if we want to add new elements to lower level sets or lists in objects like SPECTRAL_TABLE†:

```
(53.0) FOR_EACH item IN SPECTRAL_TABLE
(53.1) WHERE     item.SUBSTANCE = 'ASPIRIN'
(53.2) DO INSERT
(53.3)     ⟨|'C2H5OH', 10.0, ⟨|0, 1, ⟨⟩ |⟩ |⟩
(53.4)     INTO item.SPECTRA
(53.5) END
```

Here, a new tuple is added to the SPECTRA field of the 'ASPIRIN' entry.

In the previous examples [compare (51.1), (52.1), (53.3)], the values of the new items have been specified as user defined constants. Alternatively, these values may be specified by a query, provided the query results in a single element or in a list (set) of elements compatible with the receiving list (set).

4.3 *Update (or Assignment) Operations*

While deletion and insertion operations are applicable for sets and list only, new values may be assigned to objects of any type, e.g.

(54) L[1]: = 6 (assigns 5 to first item of L)

(55) SPECTRAL_TABLE: = {}
(removes all entries from SPECTRAL_TABLE by overwriting the contents by an empty set)

Again, the FOR_EACH construct is needed to specify objects that cannot be directly addressed. A first example increases by 10% every element of *S* which is larger than 5:

```
(56.0) FOR_EACH x IN S
(56.1) WHERE     x > 5
(56.2) DO x: =   x * 1.1 END
```

A second example shows updates of lower level elements in a non-flat table. For every tupel in the SPECTRAL_TABLE, we want to update every tuple of the SPECTRA field:

```
(57.0) FOR_EACH item IN SPECTRAL_TABLE
(57.1)   DO
(57.2)     FOR_EACH spectrum IN item.SPECTRA
(57.3)       DO
(57.4)         spectrum.S_GRAPH.START: = 0.0;
(57.5)         spectrum.S_GRAPH.STEPW: = 5.0
(57.6)       END
(57.7)   END
```

Note that the DO-END-block (57.3–6) of the inner FOR_EACH embraces two basic update operations. This block feature is especially useful, if several fields of a tuple have to be updated. Of course, such an operation sequence can be understood as another compound DMF operation.

4.4 *Keeping Intermediate Results*

Consider the subsequent assignment statement:

```
(58.0) RESULT: =
(58.1)   SELECT x
(58.2)   FROM   x IN SPECTRAL_TABLE
(58.3)   WHERE (EXISTS xx IN x.SPECTRA:
(58.4)             xx.SOLVENT = 'C2H5OH')
```

If RESULT has explicitly been declared in advance (see Section 2.2), statement (58) will be accepted if its elements (i.e. tuples) have the same type as the tuples resulting from (58.1–4).

On the other hand, RESULT needs not necessarily be declared *explicitly* since the necessary type information can be deduced from (58.1–4).

We propose to exploit these facts in the following way: If the target of an assignment has not been declared explicitly, the lifetime of the target will end with the current data base session.

This proposal complements the USE construct discussed in Section 3.7.3. While the USE feature should be used for intermediate values which are exclusively used in the scope of the USE construct, the generation of a temporary object by assignment is preferred for intermediate data to be repeatedly used in different scopes.

5. CONCLUSIONS

In this paper an SQL-like langugage for an extended NF^2 data model has been presented. It is the intention of this data model to provide appropriate data structures for applications which cannot be supported in a natural way by the classical relational approach. Obviously, those data structures require a richer repertoire of operations for manipulation. However, it can be shown [31] that the necessary extensions of traditional language concepts does not increase the complexity of simple operations on simple 1NF data structures. On the other hand, when applications require complex NF^2 structures, the corresponding operations remain manageable; when using simple structures instead, complexity of operations may even be increased [13, 31].

†For simplicity reasons, the spectral values have been specified by an empty vector "⟨⟩" in (53.3).

Contrasted with other proposals (e.g. [32, 33]) our extension of SQL can be considered as a more evolutionary approach, the potential of which has not even been fully exploited yet. For instance,

- Additional basic data types (e.g. "reference" type)
- Additional primitive operations
- More liberal syntax (e.g. mandatory use of free identifiers only in case of ambiguities)
- Generalized projection facilities using a result scheme specification (e.g. [19])
- Grouping and ordering operations with enhanced reformatting facilities
- Multilevel nesting and unnesting operations (e.g. [19])

are worthwhile for consideration. In addition to these "usability" extensions, further model extensions proper are attractive, too. The "reference" data type [8, 34], for instance, is not only a handy means to support $n:m$ relationships, but it also provides a considerable potential for model extensions, e.g. with respect to recursive (rather than nested) data structures. Of course, extensions of this type require appropriate operations (see [30, 35]), e.g. transitive closure operations, or operations avoiding excessive explicit join specifications. A further non-trivial extension would be provided by the non-atomic data type "multiset" [36]. Its merits should be obvious and need not be discussed here.

In the last few years, treatment of time in DBMS has become popular (e.g. [37]). While the extended NF^2 model offers quite comfortable facilities to handle different time aspects (e.g. lists to model histories of attribute values), an NF^2 model with *system controlled* time support is even more attractive.

A project group of the IBM Heidelberg Scientific Center has been involved for 3 years in the implementation of an experimental DBMS [38, 39], which is intended to support not only the extended NF^2 model as presented in Section 2, but also to adopt some of the ideas (e.g. time support) mentioned in the previous paragraphs.

Acknowledgements—This paper was suggested by H.-J. Schek, who also originated the development of the NF^2 model. B. Hansen and M. Hansen substantially contributed to the NF^2 language interface as described here. The authors gratefully acknowledge encouragement and helpful comments by their managers, colleagues, and friends. Thanks to A. Blaser, P. Dadam, V. Lum, F. Andersen, H. Blanken, K. Kuespert, H.-D. Werner, and J. Woodfill.

REFERENCES

[1] E. F. Codd. Further normalization of the database relational model. *Database Systems* (Edited by R. Rustin). *Courant Computer Science Symposia Series*, Vol. 6. Prentice-Hall (1972).

[2] B. Demo, A. DiLeva and P. Giolito. An entity-relationship language. *Information Systems: Theoretical and Formal Aspects. Proc. IFIP WG* 8.1 *Conf.*, Sitges (Spain), April 1985 (Edited by A. Sernadas, J. Bubenko and A. Olive), pp. 19–32. North-Holland (1985).

[3] J. D. Ullman. *Principles of Database Systems*. Pitman (1980).

[4] G. Bracchi, A. Fideli and P. Paolini. A language for a relational data base management system. *Proc. 6th Ann. Princeton Conf. on Information Science and Systems*, pp. 84–92 (March 1972).

[5] A. Makinouchi. A consideration on normal form of not-necessarily-normalized relations in the relational data model. *VLDB Proc.*, Tokyo, pp. 447–453 (1977).

[6] I. Kobayashi. An overview of the database management technology. Tech. Report TRCS-4-1, Sanno College, 1753 Kamikasuya, Isehara, Kanagawa 259-11, Japan (June 1980).

[7] R. Erbe *et al.* Integrated data analysis and management for the problem solving environment. *Inform. Systems* **5**, 273–285 (1980).

[8] R. A. Lorie and W. Plouffe. Complex objects and their use in conversational transactions. IBM Research Report RJ 3706 (November 1982).

[9] D. Luo and B. Yao. Form operation by Example, A language for office information processing. *Proc.* 1981 *SIGMOD Conf.*, pp. 212–223 (1981).

[10] N. C. Shu, V. J. Lum, F. C. Tung and C. L. Chang. Specification of forms processing and business procedures for office automation. IBM Research Report, RJ 3040 (1981).

[11] D. Tsichritzis. Form management. In *Omega Alpha* (Edited by D. Tsichritizis). University of Toronto, Technical Report CSRG-127 (March 1981).

[12] H-J. Schek. Methods for the administration of textual data in database systems. In *Information Retrieval Research* (Edited by R. N. Oddy *et al.*). Butterworths (1981).

[13] H.-J. Schek and P. Pistor. Data structures for an integrated data base management and information retrieval system. *Proc. VLDB Conf.*, Mexico (September 1982).

[14] G. Jaeschke. An algebra of power set type relations. IBM Wiss. Zentr. Heidelberg, Technical Report, TR 82.12.002 (December 1982).

[15] G. Jaeschke. Nonrecursive algebra for relations with relation valued attributes. IBM Wiss. Zentr. Heidelberg, Technical Report, TR 85.03.001 (March 1985).

[16] G. Jaeschke. Recursive algebra for relations with relation valued attributes. IBM Wiss. Zentr. Heidelberg, Technical Report, TR 85.03.002 (March 1985).

[17] H.-J. Schek and M. Scholl. An algebra for the relational model with relation-valued attributes. Technical University of Darmstadt Technical Report DVSI-1984-T1.

[18] M. A. Roth *et al.* SQL/NF: A query language for ¬1NF relational databases. Department of Computer Science, University of Texas, Austin, TR-85-19 (September 1985).

[19] N. C. Shu, B. C. Housel, R. W. Taylor, S. P. Ghosh and V. Y. Lum. EXPRESS: A data EXtraction, Processing, and REStructuring system. *TODS* **2**, (2), 134–174 (1977).

[20] B. C. Housel and N. C. Shu. A high-level data manipulation language for hierarchical data structures. *Proc. Conf. on Data Abstraction, Definition and Structure*, Salt Lake City, Utah (March 1976).

[21] N. Shu, B. C. Housel and V. Lum. CONVERT: A high level translation definition language for data conversion. *Commun. ACM* **18** (10), (1975).

[22] B. Jacobs. *Applied Database Logic II: Heterogeneous Distributed Query Processing*. Prentice-Hall (1985).

[23] D. D. Chamberlin *et al.* SEQUEL2: A unified approach to data definition, manipulation and control. *IBM J. Res. Dev.* **20**, 560–575 (1976).

[24] SQL/data system, concepts and facilities. IBM Corporation, GH 24-5013 (January 1981).
[25] G. D. Held, M. R. Stonebraker and E. Wong. INGRES: A relational data base system. *Proc. AFIPS National Computer Conf.*, Anaheim, Calif. (May 1975).
[26] F. Manola and A. Pirotte. CQLF—A query language for CODASYL type databases. *Proc. SIGMOD* 82, Orlando (June 1982).
[27] G. Schlageter, M. Rieskamp, U. Praedel and R. Unland. The network query language NOAH. *Proc. SIGMOD* 82, Orlando, pp. 104–110 (June 1982).
[28] B. Hansen, M. Hansen and P. Pistor. Formal specification of the syntax and semantics of a high level user interface to an extended NF2 data model (unpublished, 1982).
[29] A. Maier and R. A. Lorie. Implicit hierarchical joins for complex objects. IBM Research Report RJ3775, San Jose (1983).
[30] L. Gruendig and P. Pistor. Landinformatissysteme und ihre Anforderungen an Datenbankschnittstellen (Geographic information systems and their requirements regarding DB Interfaces). *Sprachen fuer Datenbanken* (*Data Base Languages* (Edited by J. W. Schmidt), pp. 61–75. *Informatik Facberichte* 72. Springer (1983).
[31] P. Pistor and R. Traunmueller. A database language for sets, lists, and tables. IBM Wiss. Zentr. Heidelberg Techn. Rep. TR 85.10.004 (October 1985).
[32] W. Lamersdorf, G. Mueller and J. W. Schmidt. Language support for office modelling. *VLDB Proc.*, Singapore, pp. 280–288 (1984).
[33] A. Albano, L. Cardello and R. Orsini. Galileo: A strongly-typed, interactive conceptual language. *ACM TODS* 10(2), 230–260 (1985).
[34] R. L. Haskin and R. A. Lorie. On extending the functions of a relational database system. *Proc. SIGMOD* 82, Orlando, pp. 207–212 (June 1982).
[35] P. Pistor, B. Hansen and M. Hansen Eine sequelartige Schnittstelle fuer das NF2 Modell (A SEQUEL like interface for the NF2 model). *Sprachen fuer Datenbanken* (*Data Base Languages*). *Informatik Fachberichte* 72 (Edited by J. W. Schmidt), pp. 134–147. Springer (1983).
[36] D. E. Knuth. *The Art of Computer Programming*, Vol. 2. Addison-Wesley (1971).
[37] V. Lum, P. Dadam, R. Erbe, J. Guenauer, P. Pistor, G. Walch, H.-D. Werner and J. Woodfill. Designing DBMS support for the time dimension. *Proc.* 1984 *SIGMOD Conf.*, June 18–21, Boston, Mass., pp. 115–130 (1984).
[38] V. Lum, P. Dadam, R. Erbe, J. Guenauer, P. Pistor, G. Walch, H.-D. Werner and J. Woodfill. Design of an integrated DBMS to support advanced applications. In *Datenbanksysteme fuer Buero, Technik und Wissenschaft, Informatik-Fachberichte* 94 (Edited by A. Blaser and P. Pistor), pp. 362–381. Springer (1985).
[39] P. Dadam *et al.* A DBMS prototype to support extended NF^2 relations: An integrated view of flat tables and hierarchies. *Proc. ACM SIGMOD Conf.*, Washington D.C. (1986).
[40] D. Bjoerner and C. B. Jones. The Vienna development method: The meta-language. *Lecture Notes in Computer Science* 61. Springer (1978).
[41] C. J. Date. *An Introduction to Database Systems.* Addison-Wesley (1981).
[42] P. Dadam, V. Lum and H.-D. Werner. Integrating time versions into a relational data base system. *Proc.* 1984 *VLDB Conf.*, August 27–31, Singapore, pp. 509–522 (1984).
[43] P. Fischer and D. van Gucht. Determining when a structure is a nested relation. *Proc. 11th Int. Conf. on Very Large Databases*, Stockholm (August 1985).
[44] M. Kraft. *Struktur und Absorbtionsspektroskopie der Steroide und Alkaloide.* Georg Thieme (1975).

APPENDIX A

Syntax Overview

For the language proposed in this paper, we present a simplified syntax here. It straightforwardly transliterates a subset of the abstract syntax as specified in [28]. The presentation emphasizes readability rather than rigor.

A1. Data Definition Facilities

⟨ddl-stmt⟩ ::= **CREATE OBJECT** ⟨obj-name⟩ ⟨type-def⟩
⟨type-def⟩ ::= ⟨basic-type⟩|⟨comp-type⟩
⟨basic-type⟩ ::= ⟨num-type⟩|**S_CHAR**|**BOOL**
⟨num-type⟩ ::= **INTEGER**|...
⟨comp-type⟩ ::= ⟨set-type⟩|⟨list-type⟩|⟨tuple-type⟩
⟨set-type⟩ ::= "{" ⟨type-def⟩ ["," "("⟨integer⟩ **[VAR]**) "}"
⟨list-type⟩ ::= ⟨g-list-type|⟩⟨char-list-type⟩
⟨char-list-type⟩ ::= **CHAR** ["(" ⟨integer⟩ **[VAR]** ")"]
⟨g-list-type⟩ ::= "⟨" ⟨type-def⟩ ["," "(" ⟨integer⟩ **[VAR]** ")"] "⟩"
⟨tuple-type⟩ ::= "⟨|" ⟨field-spec⟩ [{"," ⟨field-spec⟩}*] "|⟩"
⟨field-spec⟩ ::= [⟨field-id⟩ ":"] ⟨type-def⟩

A2. Data Manipulation Facilities

⟨dml-stmt⟩ ::= ⟨for-each⟩|⟨delete⟩|⟨update⟩|⟨insert⟩|⟨extend|⟨|block⟩
⟨insert⟩ ::= **INSERT** ⟨expr⟩ **INTO** ⟨obj-expr⟩
⟨obj-expr⟩ ::= ⟨id⟩|⟨index-expr⟩
⟨extend⟩ ::= **EXTEND** ⟨obj-expr⟩ **WITH** ⟨list-expr⟩ **(BEFORE|AFTER)**⟨num-expr⟩
⟨delete⟩ ::= **DELETE** ⟨obj-expr⟩
⟨for-each⟩ ::= **FOR_EACH**⟨free-id⟩**IN**⟨ls-expr⟩ **[WHERE**⟨bool-expr⟩]⟨dml-stmt⟩
⟨update⟩ ::= ⟨obj-expr⟩ ":=" ⟨expr⟩
⟨block⟩ ::= ⟨block-elem⟩ {";" ⟨block-elem⟩}*
⟨block-elem⟩ ::= ⟨for-each⟩|⟨delete⟩|⟨update⟩|⟨insert⟩|⟨extend⟩

A3. Query Facilities

⟨expr⟩ ::= ⟨list-expr⟩|⟨tuple-expr⟩|⟨set-expr⟩|⟨num-expr⟩|⟨char-expr⟩|⟨bool-expr⟩|⟨use-expr⟩
⟨general-expr⟩ ::= ⟨id⟩|⟨index-expr⟩|⟨query⟩|⟨group⟩|⟨apply-fct⟩
⟨index-expr⟩ ::= ⟨indexed-list-expr⟩|⟨indexed-tuple-expr⟩
⟨field-den⟩ ::= ⟨field-id⟩|⟨num-const⟩
⟨ls-expr⟩ ::= ⟨list-expr⟩|⟨set-expr⟩

A3.1 group and select expressions

⟨query⟩ ::= **SELECT** ⟨expr⟩ **FROM** ⟨from-expr-list⟩ **[WHERE** ⟨bool-expr⟩]
⟨from-expr-list⟩ ::= ⟨from-expr⟩ {"," ⟨from-expr⟩}*
⟨from-expr⟩ ::= ⟨free-id⟩ **IN** ⟨ls-expr⟩
⟨group⟩ ::= **GROUP** ⟨free-id⟩ **IN** ⟨ls-expr⟩ **BY** ⟨expr-list⟩

A3.2 List related expressions

⟨list-expr⟩ ::= ⟨m-list⟩|⟨order-by⟩|⟨cat⟩|⟨l-constr⟩|⟨sublist⟩|⟨d-cat⟩|⟨general-expr⟩
⟨m-list⟩ ::= **MLIST** "(" ⟨set-expr⟩ ")"
⟨cat⟩ ::= ⟨list-expr⟩ "||"⟨list-expr⟩
⟨d-cat⟩ ::= **DCAT** "(" ⟨list-expr⟩ ")"
⟨l-constr⟩ ::= "⟨" ⟨expr-list⟩ "⟩"|"⟨" "⟩"|"'"{⟨char-const⟩}*"'"
⟨expr-list⟩ ::= ⟨expr⟩ {"," ⟨expr⟩}*
⟨sublist⟩ ::= **SUBLIST** "(" ⟨list-expr⟩ "," ⟨start-pos⟩ "," ⟨len⟩ ")"

⟨start-pos⟩ ::= ⟨num-expr⟩
⟨len⟩ ::= ⟨num-expr⟩
⟨order-by⟩ ::= **ORDER** ⟨free-id⟩ **IN** ⟨list-expr⟩ **BY**⟨order-spec⟩{","⟨order-spec⟩}*
⟨order-spec⟩ ::= ⟨num-expr⟩ **[DESC]**
⟨indexed-list-expr⟩ ::= ⟨list-expr⟩ "[" ⟨num-expr⟩ "]"

*A*3.3 *Tuple related expressions*

⟨tuple-expr⟩ ::= ⟨t-constr⟩|⟨t-cat⟩|⟨general-expr⟩
⟨t-constr⟩ ::= "⟨|" ⟨field-expr⟩ {"," ⟨field-expr⟩}* "|⟩"
⟨field-expr⟩ ::= [⟨field-id⟩ ":"] ⟨expr⟩
⟨t-cat⟩ ::= ⟨tuple-expr⟩ "||" ⟨tuple-expr⟩
⟨indexed-tuple-expr⟩ ::= ⟨tuple-expr⟩ "." ⟨field-den⟩

*A*3.4 *Set related expressions*

⟨set-expr⟩ ::= ⟨elems⟩|⟨union⟩|⟨intersect⟩|⟨d-union⟩|⟨minus⟩|⟨s-constr⟩|⟨general-expr⟩
⟨elems⟩ ::= **ELEMS** "(" ⟨list-expr⟩ ")"
⟨union⟩ ::= ⟨set-expr⟩ **UNION** ⟨set-expr⟩
⟨intersect⟩ ::= ⟨set-expr⟩**INTERSECT**⟨set-expr⟩
⟨d-union⟩ ::= **DUNION** "(" ⟨set-expr⟩ ")"
⟨minus⟩ ::= ⟨set-expr⟩ **MINUS** ⟨set-expr⟩
⟨s-constr⟩ ::= "{" ⟨expr-list⟩ "}"|"{" "}"

*A*3.5 *Numeric and character expressions*

⟨num-expr⟩ ::= ⟨infix-num-expr⟩|⟨prefix-num-expr⟩|⟨num-cost⟩|⟨general-expr⟩
⟨infix-num-expr⟩ ::= ⟨num-expr⟩ num-op ⟨num-expr⟩
⟨num-op⟩ ::= "-"|"+"|"*"|"/"
⟨prefix-num-expr⟩ ::= "−" ⟨num-expr⟩|"+" ⟨num-expr⟩
⟨num-const⟩ ::= ⟨integer⟩|...
⟨char-expr⟩ ::= ⟨char-const⟩|⟨general-expr⟩
⟨char-const⟩ ::= **A|B|C|...**

*A*3.6 *Boolean and masking expressions*

⟨bool-expr⟩ ::= ⟨quant-b-expr⟩|⟨neg-b-expr⟩|infix-b-expr⟩|⟨primary-b-expr⟩|⟨general-expr⟩
⟨neg-b-expr⟩ ::= **NOT** ⟨bool-expr⟩
⟨infix-b-expr⟩ ::= ⟨bool-expr⟩ ⟨infix-bool-op⟩ ⟨bool-expr⟩
⟨infix-bool-op⟩ ::= **AND|OR**
⟨quant-b-expr⟩ ::= ⟨quantor⟩ ⟨from-expr-list⟩ ":" ⟨bool-expr⟩
⟨quantor⟩ ::= **FOR_ALL|EXISTS**
EXIST "(" **EXACTLY** ⟨num-const⟩ ")"
EXIST "(" **AT_LEAST** ⟨num-const⟩ ")"
|EXIST "(" **AT_MOST** ⟨num-const⟩ ")"

⟨primary-b-expr⟩ ::= ⟨set-comp⟩|⟨equality⟩|⟨nn-comp⟩|⟨bool-const⟩|⟨template-expr⟩
⟨nn-comp⟩ ::= ⟨num-expr⟩ ⟨comp-op⟩ ⟨num-expr⟩
⟨comp-op⟩ ::= ">"|"≥"|"<"|"≤"
⟨equality⟩ ::= ⟨expr⟩ ⟨eq-op⟩ ⟨expr⟩
⟨eq-op⟩ ::= "="|"≠"
⟨set-comp⟩ ::= ⟨elemof⟩|⟨subset⟩
⟨elemof⟩ ::= ⟨expr⟩**ELEMENT_OF**⟨set-expr⟩
⟨subset⟩ ::= ⟨set-expr⟩ ⟨set-comp-op⟩ ⟨set-expr⟩
⟨set-comp-op⟩ ::= **SUBSET_OF|SUPERSET_OF|P_SUBSET_OF|P_SUPERSET_OF**
⟨template-expr⟩ ::= ⟨expr⟩ **LIKE** ⟨template⟩
⟨template⟩ ::= ⟨list-template⟩|⟨set-template⟩|⟨tuple-template⟩
⟨list-template⟩ ::= "⟨" ⟨template-elem-list⟩ "⟩"|"⟨" "⟩"|"“"⟨string-template⟩"”"
⟨template-elem-list⟩ ::= ⟨template-elem⟩ {"," ⟨template-elem⟩}*
⟨template-elem⟩ ::= ⟨template⟩|⟨dontcare⟩|⟨expr⟩
⟨string-template⟩ ::= {⟨c-char⟩}*
⟨c-char⟩ ::= ⟨char-const⟩|⟨dontcare⟩
⟨set-template⟩ ::= "{" ⟨template-elem-list⟩ "}"|"{" "}"
⟨tuple-template⟩ ::= "⟨|" ⟨template-elem-list⟩ "|⟩"
⟨dontcare⟩ ::= "_"|"?"|"*"
⟨bool-const⟩ ::= **TRUE|FALSE**

*A*3.7 *Use and function expressions*

⟨use-expr⟩ ::= **USE** ⟨use-def⟩ {"," ⟨use-def⟩ }* **IN** ⟨expr⟩
⟨use-def⟩ ::= ⟨interm-fact⟩|⟨interm-dat⟩
⟨interm-dat⟩ ::= ⟨freeid⟩ " = " ⟨expr⟩
⟨interm-fct⟩ ::= ⟨userdef-fct-id⟩ "(" ⟨form-param-list⟩ ")" " = " ⟨expr⟩
⟨form-param-list⟩ ::= ⟨form-param-list⟩ {"," ⟨form-param⟩ }*
⟨form-param⟩ ::= ⟨id⟩
⟨apply-fact ::= ⟨fct-id⟩ "(" ⟨act-param⟩ { "," ⟨act-param⟩}* ")"
⟨fct-id⟩ ::= ⟨built-in⟩|⟨userdef-fct-id⟩
⟨built-in⟩ ::= **STRIP|LEN|CARD|INDL|...**
⟨userdef-fct-id⟩ ::= "not further specified"

*A*4. *Numerals and Identifiers*

⟨obj-name⟩ ::= ⟨id⟩
⟨free-id⟩ ::= ⟨id⟩
⟨field-id⟩ ::= ⟨id⟩
⟨id⟩ ::= "not further specified"
⟨integer ::= "not further specified"