DESIGNING A GENERALIZED NF2 MODEL
WITH AN SQL-TYPE LANGUAGE INTERFACE

P. Pistor, F. Andersen

IBM Wissenschaftliches Zentrum
D6900 Heidelberg, West Germany

Abstract

Because of its small set of data types, the relational model is constrained to simply structured data management tasks. For more advanced applications like engineering databases in the CAD/CAM area even a powerful extension like the NF^2 model is insufficient, yet: Important concepts like order and duplicates are not supported appropriately, and the free usage of composite items is prohibited by inherent asymmetries.

These drawbacks can be removed by a data model which supports atomic data, lists, multisets, and tuples in an orthogonal fashion. The necessary operations can be provided in an SQL-like framework. Compared with existing approaches, the expressive power could considerably be increased; nevertheless, the proposed SQL dialect has not become more complicated for comparable tasks, but more consistent and easier to understand.

1. Introduction

The classical relational model requests all data to be in first normal form (1NF) /Co72, Da81/. While this requirement considerably simplifies the data model, it is not a necessity /U180/. In the past, several proposals have been made for data models supporting non-normalized relations /BFP72, Ma77, Ko80/. Often, they arose in environments where the simplicity of the relational model was perceived as being too restrictive.

The NF^2 model /SP82/ was an attempt to provide the foundations for systems meeting these requirements. Its algebraic aspects have been addressed in numerous papers (e.g. /AB84, SS84, FV85, Jae85/). Suitable high level languages have been proposed, even independently of the NF^2 model /BFP72, SH77, SL81/; others /SP82, Jac85, RKB85/ have been developed in the spirit of SQL /CAE76, IBM81/.

Compared with the classical relational model, the NF^2 approach provides better facilities in modelling advanced applications, e.g. the complex structure of engineering design objects. Similar to the relational model, however, important aspects (e.g. vectors, matrices) are not adequately supported. In the past, requirements not covered by the basic model have often been reflected in operational systems by ad-hoc add-ons. Contrary to that approach, we propose extensions of the NF^2 model /HHP82, PHH83, PT85/ which consistently integrate further structural concepts (section 2). On this basis, desirable query operations can readily be identified (section 3), following the example of existing database interfaces and of appropriate programming languages. These functional requirements can be met by a generalized SQL interface, which is based on the orthogonality of data structures and the transitive closure of operations and expressions. In section 4 an applicative approach to DML facilities (deletion, insertion, assignment) is given. The DDL features matching the chosen data structures are presented in section 5.

During the exercise of setting up the formal definition /HHP82/ of an SQL like interface for the extended NF^2 model, the definition tool /BJ78/ in turn influenced the language to be specified. Similar effects can be observed with other database language proposals /ACO85, La84, Shp81/. While their scope is even broader, our proposal shows how far the *evolution* of existing approaches can be stretched.

2. Extended NF^2 Structures

2.1. Relations with Relation Valued Attributes

As originally proposed (e.g. /SP82/), an NF^2 relation is a set of (equally structured) tuples (or records), the fields (attributes) of which contain either "atomic" values (e.g. numbers, strings) or relations. The latter may be "flat" or NF^2 relations, in turn. Obviously, classical "flat" relations are just a special case of NF^2 relations.

Fig. 1 gives an example of an NF^2 relation with a two level hierarchy. As outlined elsewhere (e.g. /SP82/), structures like the REPORTS table are advantageous with different respects:

- Related information need not be split into different tables (see Fig. 2).
- Reports to be generated from databases often exhibit a hierarchical structure. NF^2 structures match this fact.
- NF^2 structures readily capture 1:m relationships.

2.2. Extending the Original Approach

When real world facts are to be modelled, NF^2 structures as outlined above provide a great deal of freedom not available with the classical relational model. Nevertheless, there are several drawbacks, as demonstrated below by a couple of observations. When taking a more general view of these findings, desirability of *extended* NF^2 structures becomes obvious.

2.2.1. Observations on Unary Tables

As long as the number of attributes is greater than one, tables are quite handy to model collections of

Proceedings of the Twelfth International Conference on Very Large Data Bases **Kyoto, August, 1986**

{REPORTS}				
REP_NO	<AUTHORS>	TITLE	{DESCRIPTORS}	
	NAME		KEYWORD	WEIGHT
1179	Jones,A.	Concurrency and ...	Concurrency Control	60
			Recovery	30
			Distribution	10
1189	Miller,H.	Text Editing and ...	String Search	70
	Abraham,C.		Formatting	30
	Meier,F.			
1292	Poe,A.	Branch and Bound ...	Math. Optimization	60
	Tacker,J.		Garbage Collection	40

Figure 1: Table with unary and binary table valued attributes. "Inner" tables (also referred to as "repeating groups") need not be flat as in this example; key attributes are printed in bold face. (multiset: {..}, list: <..>, tuple: [..]).

equally structured items. With unary tables, however, the tuple level layer might be undesirable. Consider for instance the AUTHORS attribute of the REPORTS table (Fig. 1). Having in hand some AUTHORS tuple, one does not have the name value itself, yet. Instead, one needs a - redundant - intermediate access to the NAME level. This situation can be avoided by a more flexible model which allows for sets in general, not only for sets of tuples.

With the AUTHORS column, another anomaly can be observed. Assume that this REPORTS table attribute contained foreign key entries representing tuples of an address table ADDRS not further to be detailed here. The question arises which of the following interpretations for an AUTHORS field is better justified:

1. A subset of the ADDRS table.
2. A set of tuples with *one* field, the content of which replicates an ADDRS tuple.

Strictly speaking, none of these interpretations is exactly mirrored by the structures provided by the strict NF^2 model. To justify the first view the keys

<AUTHORS>	
REP_NO	NAME
1179	Jones, A.
1189	Miller, H.
1189	Abraham, C.
1189	Meier, F.
1292	Poe, A.
1292	Tacker, J.

{TITLES}	
REP_NO	TITLE
1179	Concurrency and ...
1189	Text Editing and ...
1292	Branch and Bound ...

{DESCRIPTORS}		
REP_NO	KEYWORD	WEIGHT
1179	Concurrency Control	60
1179	Recovery	30
1179	Distribution	10
1189	String Search	70
1189	Formatting	30
1292	Math. Optimization	60
1292	Garbage Collection	40

Figure 2: To capture REPORTS information (see Fig. 1) in 1NF, 3 flat tables are needed.

should be contained in a plain set rather than a unary relation. View 2 seems to be more precise than the first one but causes another problem since tuple valued fields (see below) are not admissible.

2.2.2. Tuple Valued Attributes

Modern programming languages allow for records with record valued fields. In the original NF^2 model, this can be simulated by table valued attributes with the additional constraint of having the cardinality 1. The alternative approach - tuple valued attributes - is superior since it is able to *inherently* capture that consistency constraint.

2.2.3. Concept of Order

Classical relational theory knows two types of composite items, sets and tuples. It has no notion of *lists*, i.e. composite items, the elements of which are arranged in some order. Operational relational systems, however, are less puristic (see e.g. the LIKE predicate on strings /IBM81/ or the cursor concept for "long fields" /HL82/). Above all, however, they provide facilities to specify the order in which the tuples of a query result are to be returned (thus violating the closure principle).

To clean up this situation, a properly extended NF^2 model should comprise not only unordered tables (sets of tuples), but also ordered tables (lists of tuples). In addition, lists should not mandatorily be composed of tuples; instead, any other type of list elements (see Fig. 3) should be supported.

2.2.4. Set Concept Revisited

In accordance with the homogeneity of relations, lists and sets of the extended NF^2 model are homogeneous as well. Different from sets, lists are ordered, and they allow for duplicates, while sets do not. At this point, operational relational systems deviate again from pure theory, both for implementational reasons (suppression of duplicates is expensive) and application oriented requirements (suppression of duplicates may be undesirable, e.g. when performing statistics).

To make concepts more consistent, the notion of multisets /Kn71/ should be supported rather than the notion of sets. Duplicate elimination, if required, can be supported by a generic function (cf. section 3.3) which accepts both multisets and lists.

2.2.5. Keys and Surrogates

As in case of flat tables, keys serve two purposes: First, they uniquely define specific tuples of a table. Second, they establish relationships between tuples in one or more tables. The latter facility remains necessary in the NF^2 environment for dealing with n:m relationships (see e.g. /GP83/).

With non-flat tables, the concept of keys exhibits some properties not to be observed for flat tables (see also Fig. 1):

- Non flat attributes (e.g. AUTHORS) may be involved in the set of key attributes.
- Key attribute combinations may be defined at different levels (e.g. REP_NO at table level, KEYWORD at the level of the DESCRIPTORS repeating groups).
- The concepts "key" and "uniqueness" should be kept apart /SL81/.

To establish references, the usage of surrogates /Co79/ has been proposed. This proposal has both implementational and conceptual advantages /ML83/. Especially, references to tuples of table valued attributes (e.g. to DESCRIPTORS tuples as in Fig. 1) are considerably facilitated; in addition any (sub-) object can be referred to by surrogate values.

2.3. Summarizing Extended NF² Structures

In Fig. 3, an attempt is made to visualize the differences between flat relations (a), strict NF² relations (b), and the structures of the extended NF² model (c). Other than (a) and (b), the concept of extended NF² structures is completely orthogonal: any aggregate type (list, multiset, tuple) allows any aggregate component type. Thus, the model covers not only the structures of the original NF² model. It also integrates the concepts of order and duplicates, and it allows for simple or nested multisets and lists. These additional features enable one to model concepts like text, vectors, matrices etc. in a natural way.

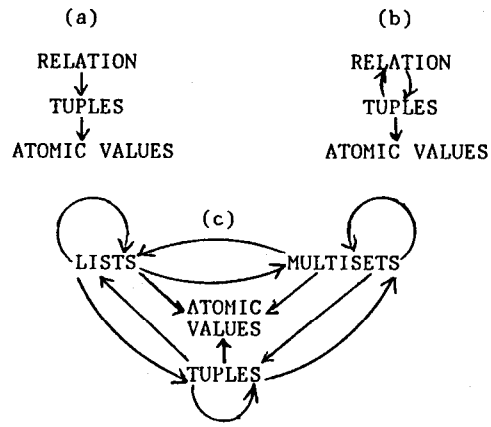

Figure 3: Structural Concepts of Different Models: (a) Classical relational model, (b) original NF² model, (c) extended NF² model. The arrows indicate possible component types. Only (a) and (b) have restrictions on admissible outermost types.

3. Query Facilities for Extended NF² Structures

The query language requirements follow quite naturally from the structures discussed in the previous section. When exploiting the syntactical resources of SQL, it is possible to

- cover these requirements to a surprisingly large extent,
- consistently integrate the new operations covering the structural extensions,
- design complementary high level constructs in the spirit of corresponding genuine SQL expressions.

Fig. 4 summarizes the operations we consider necessary for appropriately querying extended NF² structures. Many of them are an obvious consequence of the structures to be manipulated (e.g. list concatenation, counting). Since they are familiar from programming languages or from query interfaces like SQL, they are not treated here in detail, but only

Tuple-Specific Operations	
binary concatenation	
field access	(6.1)
projection	(6,27)
List-Specific Operations	
binary, multiway concatenation	
sublist extraction	
accessing elements by subscripts	
Multiset-Specific Operations	
intersection, set difference,	
binary union	/Kn71/
multiway union	/BJ78/
set restriction	(16.6)
Conversion Operations	
ordering of multisets/lists	(24,25)
lists <---> multisets	
1NF <---> NF2 structures	(17,18)
materializing references	(30)
Duplicate Suppression	
lists, multisets	(18.7)
Counting	
list, multiset elements	(24)
Arithmetic and Related Aggregation Functions	
sum, average, maximum, ...	
Comparison Operations	
lists, multisets,	
template matching ("wild cards")	
strings, numerals	(16.6-8,19.3)
Test Facilities	
quantified expressions,	
built-in predicates:	
uniqueness, undefinedness	
Explicit Constructors	
tuples, lists, multisets	(8.1,9.1,10.1)
Implicit Constructors	
lists, multisets	(13,18)

Figure 4: Query Operations Summary: Where possible, cross references to illustrative examples are given. For operations not covered in this paper, refer to /PT85, PHH83, or GP83/.

along with examples illustrating other operations, especially those related to the most central feature of SQL, the SELECT-FROM-WHERE (or "SFW") construct.

One remark on notation. When exposing the necessary SQL modifications, comparison with old style expressions are necessary at some points. To avoid confusion, they will be printed in *italics*.

3.1. Constructors

One way of understanding SQL's SFW construct is to view it as another syntactical format of an implicit set constructor /BJ78/. The analogy is given by

(1) { expr_1 | variable ε expr_2, boolean-expr }

and

(2.1) *SELECT* expr_1
(2.2) *FROM* variable expr_2
(2.3) *WHERE* boolean-expr

or by a concrete query on table DESCRIPTORS:

(3) {x.REP_NO | x ε DESCRIPTORS, x.WEIGHT > 50}

which reads in SQL as

(4.1) *SELECT x.REP_NO*
(4.2) *FROM DESCRIPTORS x*
(4.3) *WHERE x.WEIGHT > 50*

Quite intentionally, the designers of SQL did not want to stress the analogy outlined above. However, in certain cases the "predicate calculus" view considerably supports understanding. Consider e.g. the

famous query *"Which employee makes more than his manager?"*. Using a stylized payroll table

```
EMP_TAB (EMP_NO, MGR_NO, SALARY) ,
```

this query reads quite naturally as

```
(5.1) SELECT outerloop_x.EMP_NO
(5.2) FROM EMP_TAB outerloop_x
(5.3) WHERE outerloop_x.SALARY >
(5.4)           ( SELECT innerloop_y.SALARY
(5.5)             FROM EMP_TAB innerloop_y
(5.6)             WHERE outerloop_x.MGR_NO =
(5.7)                       innerloop_y.EMP_NO )
```

The usage of free variables is the key for avoiding ambiguities, not only in cases like (5), but also in processing NF^2 tables or lists and sets in general (13). Therefore, we enforce its use - at least for exposition reasons - and we stress this fact by the format of the FROM clause which binds free variables like *outerloop_x*:

```
(5.2') FROM outerloop_x IN EMP_TAB
```

So far, we have encountered not only an *implicit set constructor*, but also a rudimentary form of an *explicit tuple constructor*, namely the expression list of the SELECT clause (e.g. (2.1)). As a matter of fact, in classical SQL, "expr_1" *always* stands for a tuple constructor. With extended NF^2 structures, we need to be more explicit to avoid ambiguities. Consider the following "projection" on the REP_NO attribute of DESCRIPTORS:

```
(6.1) SELECT x.REP_NO
(6.2) FROM   x IN DESCRIPTORS
```

Does this expression denote a set of unary tuples, or is it just a plain set of numbers? As we think, the latter should be true. If we want a unary table, instead, we have to use explicit tuple constructors which are also a means to change the attribute name (details of field renaming and name inheritance are not discussed in this paper):

```
(7.1) SELECT [ NUMBER: x.REP_NO ]
(7.2) FROM   x IN DESCRIPTORS
```

Explicit tuple constructors are more than just a fall back for the degenerated unary case. Instead, they are an essential contribution to the manipulation of extended NF^2 structures. E.g., if one wants to retrieve a binary table from DESCRIPTORS, this is achieved by

```
(8.1) SELECT [ x.REP_NO, x.WEIGHT ]
(8.2) FROM   x IN DESCRIPTORS
```

If one writes (note: REP_NO and WEIGHT are both INTG):

```
(9.1) SELECT < x.REP_NO, x.WEIGHT >
(9.2) FROM   x IN DESCRIPTORS
```

then the result is a multiset of lists, each having exactly two numeric elements. Finally, a multiset of 2-element multisets is requested by:

```
(10.1) SELECT { x.REP_NO, x.WEIGHT }
(10.2) FROM   x IN DESCRIPTORS
```

Expressions like (8) are to be understood as follows: If the "input" (here: DESCRIPTORS) is a set or multiset of tuples ("unordered table"), the elements may be processed in *any* order, thus producing an unordered result. In case of an *ordered* table, however, the table elements are processed in the predefined list order, the result being a list again. In any case, no attempt is made to suppress duplicates. By these conventions, the SFW construct is generalized to a generic expression which accepts both ordered and unordered input, thus supporting both implicit list and implicit multiset construction.

The question is whether well established concepts like "joining" carry over to lists as well. We shall demonstrate it with a tiny example, which also shows that the SFW mechanism is applicable for lists or multisets composed of elements other than tuples. Let V1 and V2 be two numerical vectors:

```
(11)  V1 = < 1, 2, 5>, (12)  V2 = <10, 5, 3>
```

By (13) we obtain a list (14) of pairs such that the first element is smaller than the second one:

```
(13.1)  SELECT <x, y>
(13.2)  FROM   x IN V1, y IN V2
(13.3)  WHERE  x < y

(14)  < <1,10>, <1,5>, <1, 3>, <2,10>,
        <2, 5>, <2,3>, <5,10>          >
```

Note that the use of free variables is the key for extending the applicability of the SFW constructs to lists and sets of "non-tuple" elements. One should also note that the order of the FROM-list elements (13.2) is important, if lists are involved. If we had

```
(13.2') FROM y IN V2, x IN V1
```

instead of (13.2), the result would be different from (14), since - loosely speaking - "y" is in the outer loop, now:

```
(15)  < <1,10>, <2,10>, <5,10>, <1,5>,
        <2, 5>, <1, 3>, <2, 3>         >
```

If "V1" and "V2" were *multisets* of numbers, (13) would return a multiset of pairs. In this case, the order of the FROM-list elements ((13.2) and (13.2')) would be irrelevant (traditional symmetry of join in case of (multi-)set operands).

It is also possible to define the meaning of joins between lists and multisets. For details, the reader is referred to /PT85/.

3.2. Nested Queries

So far, we have not touched queries against typical NF^2 structures like REPORTS (see Fig. 1). As long as we need not "look into" non-atomic fields, the SFW construct - and some appropriate primitive operations - will cover most needs. However, additional facilities are required to descend into nested structures. Nested queries, no longer subject to the restrictions of classical SQL, are a major means for that end. Consider e.g. the task of extracting from REPORTS the REP_NOs together with their KEYWORDs of WEIGHT 30 (restriction and restructuring at DESCRIPTORS fields level). In the query below, this is achieved by the subquery (16.2-4). The nested structure of the intended result (including attribute renaming) is reflected by the structure of this nested query:

```
(16.0) SELECT
(16.1)  [NUM   : report.REP_NO ,
(16.2)   ITEMS: (SELECT y.KEYWORD
(16.3)           FROM   y IN report.DESCRIPTORS
(16.4)           WHERE  y.WEIGHT = 30          )]
(16.5) FROM report IN REPORTS
```

Certain DESCRIPTORS fields might not contain any tuple of the required weight (e.g. REP_NO 1292 in Fig. 1). In this case, the resulting ITEMS field would be empty. If we want to discard the associated report, an appropriate predicate on "report" is required:

```
(16.6) WHERE {} ≠ (SELECT y
(16.7)             FROM  y IN report.DESCRIPTORS
(16.8)             WHERE y.WEIGHT = 30          )
```

Syntactically, this predicate is similar to subqueries in classical SQL (except for the fact that a table valued field is acted upon).

Instead of the subquery style of (16.6-8), a quantified expression might be more attractive:

```
(16.6') WHERE EXISTS y IN report.DESCRIPTORS:
(16.7')                ( y.WEIGHT = 30 )
```

The main message of example (16) is this: The language we propose has been designed such that the user can exactly specify the intended query result. Other proposals /SH77/ trade expressive power for syntactical simplicity. As has been demonstrated /Ha80/ for hierarchical structures, the price of simplicity is ambiguity. By restricting semantics, ambiguity can be resolved in different ways; however, it is hard to decide which of them is preferable.

3.3. Nesting, Unnesting, Grouping

Compared with the classical relational algebra, two distinctive features of the NF^2 algebra /SS84, Jae85/ are the UNNEST and NEST operations. We will first show that the extended SFW construct is powerful enough to cover these operations, and then discuss the desirability of dedicated operations.

An example of unnesting is the generation of the DESCRIPTORS table contents from REPORTS; if we did so "by hand", we would repeat - for any REPORTS tuple - the required top level information as often as we have tuples in the associated DESCRIPTORS field. This is in essence a Cartesian product which shows up in (17.4-5) below:

```
(17.1) SELECT [ REP_NO   : rep.REP_NO ,
(17.2)          KEYWORD  : desc.KEYWORD,
(17.3)          WEIGHT   : desc.WEIGHT ]
(17.4) FROM   rep  IN REPORTS,
(17.5)        desc IN rep.DESCRIPTORS
```

This join operation is very efficient since one touches only once the information required for the intended result. The advantage over dedicated operations /PHH83, RKB85/ is that the full power of the SFW construct remains available.

While unnesting turns out as a special join operation, the "inverse" operation (nesting) is covered by an appropriate nested query. Consider a projection of REPORTS on the attributes REP_NO and DESCRIPTORS. By nesting, one can obtain this information from the flat DESCRIPTORS table:

```
(18.0) SELECT
(18.1) [REP_NO      : y ,
(18.2)  DESCRIPTORS:(SELECT [KEYWORD: x.KEYWORD,
(18.3)                       WEIGHT : x.WEIGHT]
(18.4)               FROM    x IN DESCRIPTORS
(18.5)               WHERE   x.REP_NO = y    ) ]
(18.6) FROM
(18.7)  y IN UNIQUE( SELECT z.REP_NO
(18.8)               FROM   z IN DESCRIPTORS )
```

In this formula, we first obtain (18.7-8) a (proper) set of REP_NOs from DESCRIPTORS. This set determines the number of non-flat tuples to be constructed in (18.1-5). While the REP_NO information (18.1,7,8) is provided by the the operation for duplicate suppression (UNIQUE), the repeating group itself is constructed by a subquery (18.2-5) directly addressing the DESCRIPTORS table.

Not regarding for this moment the question of simplicity, (18) illustrates the semantics of nesting operations. Especially, it indicates how to define nesting semantics in case of an *ordered* input table: If DESCRIPTORS were a list of tuples, then the ordering of the *top level* result tuples were determined by the semantics of the UNIQUE operation. *Within the repeating groups*, tuples would be arranged according to the relative order they had in the input table.

Like nesting, grouping also increases the number of hierarchical levels. The difference is that nesting restructures tuples, while grouping simply partitions the input, leaving the tuple structure untouched. To further stress the difference, compare (18) with the SFW expression (19), which specifies the grouping of DESCRIPTORS tuples by REP_NOs:

```
(19.0) SELECT
(19.1)  (SELECT y
(19.2)   FROM   y IN DESCRIPTORS
(19.3)   WHERE  y.REP_NO = x.REP_NO)
(19.4) FROM
(19.5)   x IN UNIQUE(SELECT z.REP_NO
(19.6)               FROM   z IN DESCRIPTORS)
```

Like nesting, grouping can be applied both on ordered and on unordered input. Evidence is given by (19), where DESCRIPTORS might equally well be a multiset or a list of tuples. Details can be deduced from (19) in analogy to the discussion on nesting.

Examples (18) and (19) convincingly illustrate the expressive power of the SFW construct. Nevertheless, this construct should be complemented by dedicated GROUP and NEST operations for different reasons (compact notation, ease of use, query optimization). Following /RKB85/, nesting operations can be considerably boiled down when restricted to tables as in (18):

```
(20.1) NEST DESCRIPTORS
(20.2) ON   (KEYWORD, WEIGHT) AS DESCRIPTORS
```

Having a GROUPing facility, (19) is rewritten as follows:

```
(21.1) GROUP
(21.2)       x IN DESCRIPTORS
(21.3) BY    x.REP_NO
```

Other than in (20), we use free variables here. Therefore, GROUPing is *not* restricted to tables, but may operate any list or multiset. E.g., the set

```
(22)  MS = { 3, 1, 2, 5, 6 }
```

of numbers may be partitioned into odd and even equivalence classes (i.e. { {1,5,3}, {2,6} }) as follows:

```
(23.1) GROUP
(23.2)       x IN MS
(23.3) BY    (x MOD 2)
```

3.4. Ordering

ORDERing is a generic operation which takes multisets or lists and returns a list, the elements of which are sequenced according to the order specification. Unlike NESTing and GROUPing, the ORDERing operation is beyond the possibilities of the SFW construct.

The syntax of the ORDERing operation is designed in analogy to GROUPing. In doing so, we are able to order not only tables (24), but also multisets (25) or lists of "anonymous" elements:

```
(24) ORDER x IN REPORTS BY COUNT( x.AUTHORS )

(25) ORDER x IN MS BY (x MOD 3)
```

3.5. Projection

It is the virtue of the SFW construct to allow for projection, restriction, and renaming within one operation. In case of *plain* projection, however, the operation is overpowered. Consider for instance the following projection on the attributes REP_NO, DESCRIPTORS, and KEYWORD:

```
(26.1) SELECT
(26.2) [REP_NO      :  x.REP_NO                    ,
(26.3)  DESCRIPTORS: (SELECT [y.KEYWORD]
(26.4)                FROM y IN x.DESCRIPTORS)]
(26.3) FROM x IN REPORTS
```

Similar to NESTing (20) it would be sufficient, instead, to indicate the involved attributes /SH77/:

```
(27)   REPORTS•(REP_NO, DESCRIPTORS(KEYWORD))
```

Here, the right operand of the projection operator "•" is a template denoting a subschema of the left operand, which in turn may be a list/multiset of tuples. The left operand could also be a *single* tuple. In this case, "•" would return a tuple rather than a table.

3.6. Joins via Surrogates

Let REPORTS_S be a table similar to REPORTS (Fig. 1) but having an AUTHORS field with surrogates referring to tuples of an address

```
(28) ADDRS(FIRST_NAME,LAST_NAME, ....)
```

Based on these tables, a table of author lists can be obtained by the following join operation:

```
(29.0) SELECT
(29.1) [REP_NO : x.REP_NO,
(29.2)  AUTHORS:(SELECT
(29.3)           [FIRST_NAME: y.FIRST_NAME,
(29.4)            LAST_NAME : y.LAST_NAME ]
(29.5)           FROM z IN x.AUTHORS, y IN ADDRS
(29.6)           WHERE SURROGATE(y) = z )        ]
(29.7) FROM   x IN REPORTS_S
```

In this expression, the explicit naming of the ADDRS table is redundant since surrogate values (here: z) uniquely identify data objects (here: ADDRS tuples).

The join (29.5-6) can be condensed (further compaction were achievable by the operation "•" of sect. 3.5) by a generic function "MAT" which takes a scalar surrogate or a list/multiset of surrogates and returns the associated object or the list/multiset of objects, resp.:

```
(30.0) SELECT
(30.1) [REP_NO :  x.REP_NO,
(30.2)  AUTHORS: (SELECT
(30.3)            [FIRST_NAME: y.FIRST_NAME,
(30.4)             LAST_NAME : y.LAST_NAME  ]
(30.5)           FROM y IN MAT(x.AUTHORS)    ]
(30.6) FROM   x IN REPORTS_S
```

Note that (29) is robust against unresolvable surrogate values. To preserve this property in (30), MAT must be defined such that it ignores dangling references when taking a list/multiset of surrogate values.

4. DML Facilities

To change data in database relations, SQL provides three basic DML sections:

- deletion of tuples,
- insertion of a single tuple or a set of tuples,
- overwriting contents of (atomic) tuple fields.

When considering REPORTS as a typical NF^2 structure, it becomes obvious immediately that these operations must be generalized to support

- overwriting contents of a repeating group (e.g. an AUTHORS field),
- inserting additional tuples into a repeating group (e.g. DESCRIPTORS field),
- removing tuples from a repeating group (e.g. an AUTHORS field).

To support these requirements, facilities are needed to identify the specific repeating groups which are to be operated upon. Most probably two complementary approaches will be needed to provide this facility. One of them (e.g. /PHH83,PT85/) is strongly influenced by the imperative style of conventional programming languages. In this paper (see examples below) a more declarative approach is presented which attempts to preserve the spirit of SQL. As we will see, the operations are provided in such a format that one can appropriately process not only tables (both ordered and unordered), but also lists, multisets and tuples in general.

4.1. Deletion

As an example for deleting tuples from a repeating group, consider *"Remove all DESCRIPTORS entries with WEIGHT < 50"*:

```
(31.1) DELETE  y
(31.2) FROM    y IN x.DESCRIPTORS, x IN REPORTS
(31.3) WHERE   y.WEIGHT < 50
```

As illustrated above, free variables enable one to descend in the REPORTS table. In the same way, one can descend in anonymous structures like LL (see (14)). E.g., the following statement deletes certain elements of inner lists:

```
(32.1)  DELETE y  FROM y in x, x in LL
(32.2)  WHERE (y < 4) AND (SUM(x) < 10)
```

4.2. Insertion

Like deletion, insertion is restricted to lists and multisets. First, we demonstrate the insertion of a tuple into an ordered repeating group:

```
(33.1)  INSERT [ NAME: 'Duck, D.' ]  AFTER  1
(33.2)  INTO   x.AUTHORS  FROM x IN REPORTS
(33.3)  WHERE  EXISTS y IN x.AUTHORS:
(33.6)                      (y.NAME = 'Ente, G.')
```

Instead of explicitly enumerated tuples (33.1), any type compatible expression denoting a target compatible list or multiset may be used for insertion:

```
(34.1)  INSERT (SELECT y*(KEYWORD,WEIGHT)
(34.2)          FROM   y IN DESCRIPTORS
(34.3)          WHERE  y.REP_NO=x.REP_NO )
(34.4)  INTO   x.DESCRIPTORS
(34.5)  FROM   x IN REPORTS
```

Of course, insertion can be supported not only for tables, but also for lists and sets in general.

4.3. Assignment

Other than /RKB85/, insertion and deletion is not understood as an update operation. Instead, updating is conceived as replacing the contents of data base objects of any type. In (35) e.g. a specific DESCRIPTORS field in REPORTS is overwritten by a new tuple:

```
(35.1) ASSIGN {[KEYWORD:'B tree', WEIGHT:50]}
(35.2) TO     x.DESCRIPTORS
(35.3) FROM   x IN REPORTS
(35.4) WHERE  x.REP_NO = 205
```

Examples involving other types of source or target expressions are omitted here. They are analogous to the DML examples given so far.

5. DDL Language

In systems based on the classical relational model, it is in essence sufficient to specify the attribute names, the data types to be associated with them, and, of course, the relation name. With NF^2 structures, aggregate types cannot be defaulted but must be specified explicitly. A possible minimal set of declaration facilities requires some predefined basic data types, and type constructors for multisets ({..}), lists (<..>), and tuples ([..]). Using these facilities, it is possible to declare objects like a list of integers (36)

```
(36.1) CREATE L_I < INTG > END
```

as well as our REPORTS table (37):

```
(37.1) CREATE REPORTS
(37.2) {[ REP_NO     : INTG                          ,
(37.3)    AUTHORS    : <[ NAME: CHARVAR (30) ]>,
(37.4)    TITLE      : CHARVAR (250)                 ,
(37.5)    DESCRIPTORS: {[ KEYWORD: CHARVAR(50),
(37.6)                    WEIGHT : INTG       ]}
(37.7) ]} END
```

If undefined values are admissible (e.g. undefined AUTHORS values), this must be explicitly allowed (NULLS option).

The facilities discussed so far do not cover keys or surrogates. Key attributes or key attribute combinations may be specified by clauses like

```
(37.6')  ...] KEYS (KEYWORD) }
(37.7')  ...] KEYS (REP_NO; AUTHORS, TITLE) } END
```

(37.6') defines the key attribute of a repeating group. In (37.7') two alternative "top level" keys are specified. Especially interesting is the combination AUTHORS/TITLE, since it involves a non-atomic attribute. For obvious reasons, a NULLS option is incompatible with key attributes.

If a DBMS is expected to support referential integrity, it is necessary that data representing foreign keys need to be indicated as such. When using surrogates, the system is first of all explicitly told where surrogates are to be generated. Examples:

```
(37.7'')        ...    ]  SURROGATED }

(38)   CREATE LL <<INTG>  SURROGATED> END
```

(In the latter example, the surrogates are intended to identify the inner lists of "LL". In contrast, ordinary keys are restricted to tuples).

Surrogate values must be strictly type-bound. Otherwise, expressions like (30) would return inhomogeneous multisets, thus violating the closure principle. Type binding can be specified directly (e.g. "REF <INTG>"), or may be further restricted to certain objects (e.g. for authorization):

```
(39.1)  ... { REF y IN x.DESCRIPTORS,
(39.2)              x IN REPORTS        ) ...
```

Even mutually recursive references are allowed, provided they are declared within one CREATE operation:

```
(40.0) CREATE OBJECT
(40.1)   REF_OBJ1 {[ F1  : REF x IN REF_OBJ2,
(40.2)              VAL : INTG                ]},
(40.2)   REF_OBJ2 {[ F2  : REF y IN REF_OBJ1,
(40.2)              AUTHOR: CHARVAR (20)     ]}
(40.3) END
```

6. Conclusions

As has been pointed out by several authors, the relational model is too poor to adequately support advanced applications like engineering databases. Approaches like the NF^2 model provide more freedom in capturing complex facts, but do not consistently support important concepts like order and duplicates, and instead of a general notion of sets, it only allows for sets of tuples. These drawbacks can be removed by a structural concept which supports atomic data, homogeneous aggregates (lists, multisets) and inhomogeneous aggregates (tuples), where any supported data type is admissible for aggregate components. Both classical relations and strict NF^2 tables are special cases of this generalized approach.

For the manipulation of these structures we propose a language interface that provides any reasonable mapping between the supported data types. As demonstrated in this paper, such a language interface can be built on the basis of SQL. The goal was achieved by

- extending the domain of the SFW construct,
- complementing it by similarly powerful operations (e.g. grouping, ordering),
- extending the set of more basic operations (access by surrogates, subscripts etc.),
- observing the closure principle /Da84/.

In spite of increased power, complexity of the language remains moderate. Where comparable with original SQL, it has become more transparent and better to understand. While concentrating on query language aspects, we have also demonstrated that our proposal provides considerable freedom for changing stored data. We feel that it largely covers the requirements of nonstandard applications, at the same time providing an interesting migration path for hierarchical, network, and classical relational database systems.

7. References

AB84 S.Abitoul, N.Bidoit: Non First Normal Form Relations: An Algebra Allowing Data Restructuring. Rapports de Recherche No 347, Institut de Recherche en Informatique et en Automatique, Rocquencourt, France, Nov. 1984.

ACO85 A.Albano, L.Cardello, R.Orsini: Galileo: A Strongly-Typed, Interactive Conceptual Language, ACM TODS, Vol. 10(2) (1985), pp. 230-260.

BFP72 G.Bracchi et al: A Language for a Relational Data Base Management System. Proc. Sixth Ann. Princeton Conf. on Information Science and Systems, Mar. 1972, pp. 84-92.

BJ78 D.Bjoerner, C.B.Jones: The Vienna Development Method: The Meta-Language. Lect. Notes in Comp. Science 61, Springer, 1978.

CAE76 D.D.Chamberlin et al.: SEQUEL2: A Unified Approach to Data Definition, Manipulation and Control, IBM Journ. Res. Devel. 20 (1976), pp. 560-575.

Co72 E.F.Codd: Further Normalization of the Database Relational Model. Database Systems, ed. R. Rustin, Courant Comp. Sc. Symposia Ser. Vol. 6, Englewood Cliffs, N.Y. Prentice Hall, 1972.

Co79 E.F.Codd: Extending the Data Base Relational Model to Capture More Meaning. ACM TODS, Vol. 4(4) (1979), pp. 379-434.

Da81 C.J.Date: An Introduction to Database Systems, Addison-Wesley Publishing Company, 1981.

Da84 C.J.Date: Some Principles of Good Language Design with Special Reference to the Design of Database Languages. ACM SIGMOD Record 14(3) (Nov. 1984), 1-7.

FV85 P.Fischer, D.van Gucht: Determining when a Structure is a Nested Relation. Proc. Eleventh Intern. Conf. on Very Large Databases, Stockholm, (Aug. 1985).

GP83 L.Gruendig, P.Pistor: Landinformationssysteme und ihre Anforderungen an Datenbankschnittstellen. In /Sc83/, pp.61-75.

Ha80 T.H. Hardgrave: Ambiguity in Processing Boolean Queries on TDMS Tree Structures: A Survey on Four Different Philosophies. IEEE Trans. Softw. Engin. 6(4), July 1980.

HHP82 B.Hansen, M.Hansen, P.Pistor: Formal Specification of the Syntax and Semantics of a High Level User Interface to an Extended NF2 Data Model (unpublished, 1982).

HL82 R.L.Haskin, R.A.Lorie: On Extending the Functions of a Relational Database System. Proc. SIGMOD 82, Orlando, June 1982, pp. 207-212.

IBM81 SQL/Data System, Concepts and Facilities, IBM Corporation, GH 24-5013, Jan. 1981.

Jac85 B.Jacobs: Applied Database Logic II: Heterogeneous Distributed Query Processing. Prentice-Hall, Englewood Cliffs, 1985.

Jae85 G.Jaeschke: Recursive Algebra for Relations with Relation Valued Attributes. IBM Wiss. Zentr. Heidelberg Techn. Rep. TR 85.03.002, March 1985.

Kn71 D. E. Knuth: The Art of Computer Programming, Vol. 2. Addison Wesley Publishing Company, Reading (Mass), 1971.

Ko80 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.

La84 W.Lamersdorf et al: Language Support for Office Modelling. VLDB Proc., Singapore, 1984, pp. 280-288.

Lo84 R.A.Lorie et al: User Interface and Access Techniques for Engineering Databases. IBM Res. Rep. RJ 4155 (45943), Jan. 1984.

Ma77 A. Makinouchi: A Consideration on Normal Form of Not-Necessarily-Normalized Relations in the Relational Data Model. VLDB Proc., Tokio, 1977, pp. 447-453.

ML83 A.Maier, R.A.Lorie: Implicit Hierarch. Joins for Complex Objects. IBM Res. Rep. RJ3775, 1983.

PHH83 P.Pistor, B.Hansen, M.Hansen: Eine sequelartige Schnittstelle fuer das NF2 Modell. In /Sc83/, pp. 134-147.

PT85 P.Pistor, R.Traunmueller: A Database Language for Sets, Lists, and Tables. IBM Wiss. Zentr. Heidelberg Techn. Rep. TR 85.10.004, Oct. 1985.

RKB85 M.A. Roth et al: SQL/NF: A Query Language for ¬NF Relational Databases. Deptm. Comp. Scienc. Univ. of Texas, Austin, TR-85-19, Sept. 1985.

Sc83 J.W.Schmidt (ed.): Sprachen fuer Datenbanken. Informatik Fachberichte 72, Springer Verlag, Berlin-Heidelberg-New York, 1983.

Shp81 D.W.Shipman: The Functional Data Model and the Data Language DAPLEX. ACM TODS, Vol. 6(1) (1981), pp. 140-173.

SH77 N.C.Shu et al.: EXPRESS: A Data EXtraction, Processing, and REStructuring System, TODS 2(2) (1977), pp. 134-174.

SL81 N.C.Shu, V.Y.Lum et al: Specification of Forms Processing and Business Procedures for Office Automation, IBM Res. Rep. RJ 3040, 1981.

SP82 H.-J.Schek, P.Pistor: Data Structures for an Integrated Data Base Management and Information Retrieval System, Proc. VLDB Conf. Mexico, Sept. 1982.

SS84 H.-J.Schek, M.Scholl: An Algebra for the Relational Model with Relation-Valued Attributes. TR DSVI-1984-T1, Techn. Univ. Darmstadt, 1984.

Ul80 J. D. Ullman, Principles of Database Systems, Pitman, London, 1980.