Sorting, Grouping and Duplicate Elimination in the Advanced Information Management Prototype

G. Saake
Technische Universität Braunschweig, Abteilung Datenbanken, D-3300 Braunschweig, West Germany

V. Linnemann, P. Pistor
IBM Scientific Center Heidelberg, Tiergartenstrasse 15, D-6900 Heidelberg, West Germany

L. Wegner
Universität Kassel, D-3500 Kassel, West Germany

Abstract

Sorting, duplicate suppression and grouping are important operations in relational database management systems. This paper is devoted to the related language features and their implementation in the Advanced Information Management Prototype AIM-P. The query language HDBL is an SQL-like database language supporting the extended NF² data model. The proposed language extensions follow the classical SQL approach for sorting and duplicate elimination by extending the SFW construct with appropriate clauses. For the grouping operation we chose a new syntactical construct because the implicit structure transformation of grouping differs from the sorting and duplicate suppression operations. Finally, the integration into the query evaluation of the AIM prototype is described.

1. Introduction

Traditionally, sorting subsystems have played an important role in data base management systems. In relational systems like DB2, for example, they are the basis for producing sorted output, for eliminating duplicates, and for operations requiring an intermediate partitioning of input tables. They provide additional facilities for optimizing join operations and can be extremely advantageous for the creation of indexes.

This paper is devoted to the definition and implementation of sorting, grouping, and duplicate suppression in AIM-P /DaK86,ALPS86,LKD88/, a database system supporting an extended NF² data model. Some introductory information about the underlying data model is given in section 2. It illustrates how basic operations of the NF² algebra can be packaged in an SQL-like fashion, how their semantics are to be generalized to cope with the structures supported in AIM-P, and how they are related to sorting, grouping, and duplicate elimination.

With sorting (sect. 3) and duplicate elimination (sect. 4) we have chosen the classical SQL approach of providing these facilities via clauses added to the SFW construct. In comparison to initial austere proposals /PA86,PT86/, this approach allows for an intuitive and compact notation; in addition, this syntactical format seems to be better suited for translation into optimized sequences of lower level operations. These goals have also guided the design of the grouping operation (sect. 5). In this case however we had good reasons for developing syntactical solutions which deviate from classical SQL. Section 6 describes an evaluation machine which is able to exploit the facilities of an appropriately designed sorting subsystem in realizing these functions.

2. Extended NF² Data Model

2.1 Supported Structures

The eNF² data model of AIM-P /PT86,PA86,Pi87/ is based on the idea of non first normal form relations /AB84,SS84/. In terms of the extensions, the two models can be contrasted as follows:

- Top level objects are not necessarily relations. Scalars or tuples are admissible as well.
- The notion of relations is generalized to the notion of tables, i.e. collections of tuples.
- Collections may be ordered (lists) or unordered (sets or multisets). Collections may be free of duplicates ("unique") or not.

The work was done while the first author was at the IBM Scientific Center in Heidelberg on leave from Technische Universität Braunschweig

Proceedings of the Fifteenth International
Conference on Very Large Data Bases

Amsterdam, 1989

- Collections need not necessarily be composed of tuples. Sets of integers or lists of tables are conceivable as well.
- Like the strict NF² model, tuples need not be composed of "atomic" items. Besides tables, however, fields may also contain tuples, or collections other than tables.

The following figure shows an eNF² structure which will also be used in the following chapters as an accompanying basic structure for language examples.

{ Literature }			
Title	< Authors >		{ Keywords }
	Surname	Firstname	

Designing a Generalized NF2 Model with an SQL-Type Interface	Pistor	Peter	Database Systems
	Andersen	Flemming	Nested Relations
			Query Languages
....			

The eNF² relation Literature is a multiset of tuples having two non-atomic attributes, namely an ordered list Authors and a multiset Keywords. In contrast to 'puritanical' relational and NF² database theory, the notions of sorting and duplicate elimination operators are well defined in the extended NF² data model because of the explicit handling of lists and multisets.

- The property 'is sorted' can only be stated for collections of type list, because for sets and multisets the ordering is irrelevant per definition.
- For collections other than lists, the property 'free of duplicates' makes only sense in data models which distinguish semantically between sets and multisets.
- The grouping operator involves nested structures, and therefore leaves the scope of 'flat' relational theory.

In contrast to the pure NF² data model, collections with duplicates are correct eNF² data structures. For this reason, duplicate suppression in AIM-P is exclusively done on demand in contrast to /KF88/ where the demand for duplicate suppression results implicitly from the semantics of the underlying pure NF² data model.

2.2 HDBL Language

For the eNF² setting, the AIM prototype provides an SQL-like language interface (Heidelberg Data Base Language HDBL). The concepts of this language /PHH83,PA86,PT86/ need not be outlined here in detail. Instead, a few comments and illustrative examples are sufficient to provide the background for the subsequent discussions.

As in other languages of the SQL family, the SELECT ... FROM ... WHERE construct (SFW construct) plays a central role in HDBL. It is an expression which maps collections into collections. Its general format is

```
(1.1) SELECT f(x1,..., xn)
(1.2) FROM   x1 IN Input1,..., xn IN Inputn
(1.3) WHERE  p(x1,..., xn)
```

The semantics of (1) can be understood quite naturally by analogous expressions of constructive set theory, viz.

```
(1')   {f(x1,..., xn) | x1 ∈ Input1 and ... and
                        xn ∈ Inputn and p(x1,..., xn)  }
```

The term f(x1,..., xn) stands for an arbitrary expression, usually an expression returning a tuple; it provides the elements of the query result. The expression p(x1,..., xn) denotes a Boolean expression which controls whether a specific combination of elements (x1,..., xn) from the input collections Input1 through Inputn is rejected or not. The resulting collection is a list if at least one input collection is a list. In general, the result is finally assumed to be non-unique (i.e. the elements are not necessarily distinct from each other).

Expression f in (1.1) may be composed of any legal expressions of HDBL, especially SFW expressions. This is illustrated in the subsequent example:

```
(2.1) SELECT [FirstAuthor:  x.Authors[1],
(2.2)        ShortKeys  : (SELECT y FROM y IN x.Keywords
(2.3)                      WHERE  LENGTH(y) < 10)]
(2.4) FROM   x IN Literature
```

As this example demonstrates, the facilities of the SELECT clause in HDBL are far beyond plain projection in SQL for a flat setting (for similar facilities in NF² algebra see /SS84/). Rather than picking up the attributes Authors and Keywords, query (2) retrieves only the first author and short keywords.

Restriction, projection, and join are typical operations of relational algebra. HDBL supports them by SFW just like other SQL languages. In addition, SFW covers the typical NF² operations "nesting" and "unnesting". E.g.

```
(3.1) SELECT [x.Title, y.Surname, y.Firstname]
(3.2) FROM   x IN Literature, y IN Authors
```

provides Title and Authors information in an unnested ("flat") fashion. Nesting is demonstrated in the example (4) below. This example also gives further evidence that HDBL is not restricted to the table paradigm. The query

```
(4.1) SELECT [Rest      : x,
(4.2)         Partition: (SELECT y FROM y IN L
(4.3)                     WHERE x = y MOD 3)]
(4.4) FROM   x IN <0, 1, 2>
```

performs a modulo 3 partitioning of an integer list such that the partitions are displayed along with the grouping criterion.

The reader will have observed the key to this solution: the sequence (0, 1, 2) governing the partitioning is known in advance. A more general solution requires that a unique list (or set) of grouping values is extracted from the collections to be partitioned. In our case, (4) needs to be replaced by something like

```
(5.1) SELECT [Rest      : x,
(5.2)         Partition: (SELECT y FROM y IN L
(5.4)                     WHERE x = y MOD 3)]
(5.5) FROM   x IN MAKE_UNIQUE(SELECT z MOD 3 FROM z IN L )
```

Sofar we have addressed a wide range of operations that can be attributed to SFW expressions in a natural and intuitive fashion. With ordering we have an operation which is not covered by an interpretation of SFW as indicated in the analogy between (1) and (1'). Appropriate operations /PA86,RKB87/ like

```
(6)   ORDER /* every */ x IN L BY x MOD 3
```

are quite in the spirit of /Da84/: If a query result is to be ordered, this can be achieved by functional composition as in

```
(7)   ORDER x IN (SELECT ... FROM ... WHERE ...) BY ...
```

There might be situations where it is desirable to split operations into parts in order to control the sequence in which the parts are executed. On the other hand, there are good reasons for having compound operations for frequently used compositions. For example, users are quite happy to write

```
(8.1)  SELECT [x.Authors] FROM x IN Literature
(8.2)  WHERE  x.Title CONTAINS 'Darwin'
```

combining selection and projection in a dense notation rather than to write (quite correctly, by the way)

```
(9.1) SELECT [x.Authors]
(9.2) FROM   x IN (SELECT y FROM y IN Literature
(9.3)              WHERE y.Title CONTAINS 'Darwin')
```

For that reason, the designers of SQL /CAE76, ISO86/ have overloaded the SFW construct by optional clauses for ordering, grouping, and duplicate suppression. We adopt this approach with ordering (sect. 3) and duplicate elimination (sect. 4). With grouping (sect. 5), however, we believe that a dedicated operation is justified.

3. Sorting

3.1 Ordering Relation

Sorting is an operation which takes collections and re-arranges them into a list which is ordered according to given sort criteria. In conventional DBMS simple sorting criteria (i.e. expressions yielding basic type results) are sufficient, since the objects to be sorted have a simple structure (usually flat records). In contrast, HDBL is orthogonal in that it provides ordering relations for any admissible eNF² value, be it atomic, a tuple, a set or a list. We have an intuitive, well defined ordering relation on tuples (proceeding the tuple fields from left to right) as well as on lists (using the lexicographic order known from text strings). An ordering relation between set or multi-set values is less obvious. E.g., is the set {1,5} less than {2,3,4} or not ?

A discussion on different choices for generalized ordering relations is presented in /KSW89/. The most preferable one defines an ordering relation on sets and multisets according to 'cardinality first, then iteratively comparing the minimum'.

3.2 Justification of SQL-like Sorting

As already indicated in section 2.2, a less austere facility for sorting is desirable. Of course, one could enrich the ORDER operation (see (6) or (7)) by facilities for rejecting input elements and for restructuring output elements. However, this would essentially be a duplication of SFW facilities. The SQL design - an optional order clause - is probably much more clever. Before giving evidence for this allegation, let's have a first impression of SQL-like sorting mechanisms within HDBL. The subsequent query gives an example of a seemingly simple ordering criterion. It lists all the `Literature` entries ordered by their titles. Rearrangement of `Literature` is based on the lexicographic ordering relation of strings.

```
(12.1)  SELECT l FROM l IN Literature
(12.2)  ORDER BY l.Title
```

The next query delivers *all* author entries in alphabetic order. Please note that both unnesting and ordering operations are expressed by a single SFW expression, and a complete tuple (see `a` in (13.2)) is used as sort criterion.

```
(13.1)  SELECT a FROM l IN Literature, a IN l.Authors
(13.2)  ORDER  BY a DESC
```

It should be noted that - *different from SQL* - the information on sort criteria need not be contained in the final output. This is illustrated in (14) which sorts the titles of the `Literature` table according to the number of authors:

```
(14.1)  SELECT l.Title FROM l IN Literature
(14.2)  ORDER  BY COUNT(l.Authors)
```

The SQL-like approach is advantageous for several reasons: First, it puts HDBL closer to established SQL standards /ISO86/. Second, a large class of queries can be formulated in a *more concise* fashion. For example, having nothing but austere sorting facilities, example (14) were to be rewritten as

```
(15.1)  SELECT l.Title
(15.2)  FROM   l IN (ORDER  l' IN Literature
(15.3)               BY COUNT(l'.Authors))
```

or even worse:

```
(16.1)  SELECT l.Title
(16.2)  FROM   l IN
(16.3)     (ORDER l' IN
(16.4)           (SELECT [l".Title,
(16.5)                    SoMany: COUNT(l".Authors)]
(16.6)            FROM    l" IN Literature    )
(16.7)      BY l'.SoMany        )
```

It is not verbosity which is criticized here. It is the users' temptation to adopt the role of an optimizer, i.e. taking care of issues like

- compressing data before ordering (see (16.4-6)),
- arranging the sequence of SFW and ORDER operations according to whether order criteria are contained in the final output or not.

The preceeding arguments should not be understood as a case against subjecting the result of ordering operations to other operations which are applicable to lists, as in (17) below:

```
(17.1) SELECT [l.Title,
(17.2)        LexicographicallyFirstAuthor :
(17.3)                  (SELECT a FROM a IN l.Authors
(17.4)                   ORDER  BY a  ) [1] ]
(17.5) FROM   l IN Literature
```

3.3 Syntax and Semantics

The syntactical notation of the ordering clause is similar to the standard SQL proposal /ISO86/. A list of order criteria is added to the known SFW construct. The following rules in BNF syntax define the syntactical notation.

```
<SFW> :: SELECT <expr> FROM <from-list> [WHERE <expr>]
         [ORDER BY <order-expr> {"," <order-expr>}*]
<order-expr> :: <expr> [ ASC | DESC ]
```

An order expression is - in contrast to standard SQL - an arbitrary expression together with the ordering orientation (ascending or descending order). Scope and binding of variables in this expression are identical to scope and binding of the variables in the SELECT and WHERE clause of the SFW construct and are determined by the FROM list.

The following examples show ordering expressions with more complex expressions. Example (18) orders a list of points in the plane according to their distance to the origin.

```
(18.1) SELECT p FROM p IN Points
(18.2) ORDER BY p.x * p.x + p.y * p.y
```

Example (19) orders the authors by the number of their papers in descending order.

```
(19.1) SELECT a FROM l IN Literature, a IN l.Authors
(19.2) ORDER BY
(19.3)   COUNT( SELECT l' FROM l' in Literature
(19.4)          WHERE a ELEMENT_OF l'.Authors) DESC
```

As mentioned in chapter 3.1, we allow expressions yielding arbitrary eNF² type results as order criteria. As an example, query (20) orders the literature according to the set of authors derived from the author list.

```
(20.1) SELECT l.Title FROM l IN Literature
(20.2) ORDER BY MAKE_SET(l.Authors)
```

The semantics of an ordering expression does not seem to be complicated: The result of the SFW query is transformed into a list ordered by the specified sort criteria; this effect can be achieved by a *mental* evaluation model containing the following steps:

1. In a first step the cross product of all sets / lists occurring in the FROM-list is constructed (under consideration of the hierarchical variable binding). If lists are involved, the relative order of list elements is retained by following the rules given in /PT86,Da88/. If the FROM-list defines a join between two (or more) lists, nested loops may serve as a mental model for the construction of the relative order of the cross product lines.

   For the following steps, the variables are bound to the corresponding parts of the cross product lines.
2. All lines of the cross product are removed where the WHERE clause is evaluated to FALSE.
3. The sort expressions are evaluated for each remaining line.
4. The lines are sorted according to the sort criteria values computed in step 3.
5. The result is constructed out of the cross product lines as defined in the SELECT clause. This may imply a loss of the information needed for the preceding sort step.

The separation of the sort criteria computation phase (step 3) and the value comparisons (step 4) is necessary in this semantics definition, because HDBL expressions may have a nondeterministic semantics. An example is the conversion of a multiset into a list, where the order of the list elements can be arbitrarily chosen.

Before we finish the chapter and turn to the problem of duplicate elimination, we want to state a few requirements for sort algorithms to be used. Of course, the algorithms should be fast - but there are additional interesting requirements not appearing in the 'flat' case. The ordering relation on arbitrary extended NF² structures is recursively defined, so sorting on substructures may become a significant part of the total computation costs. Good sorting algorithms have to reduce these costs to a minimum by only partially sorting of substructures as proposed in /KSW89/.

4. Duplicate Elimination

4.1 Identification of Duplicates

The need for identification and deletion of duplicates arises in all data models offering sets and multisets as structuring facilities. On the one hand, set type structures should be free of duplicates per definition, and type conversion operations have to obey this property. On the other hand, the (relational algebra) operation 'projection' delivers usually results containing duplicates (not in relational

theory, but in implemented database systems). Here we concentrate on explicit query features suppressing duplicates, and do not discuss implicit duplicate elimination due to set or uni-list type database structures.

What does 'free of duplicates' for extended NF^2 structures mean ? This property can only be stated for sets and lists, and there the intuitive meaning is 'containing no identical elements'. If we fix the term 'identical' as *representation independent equality* of NF^2 objects, we see that this requires a recursively defined equality on complex NF^2 structures, which has to take special care of set type structures. The formal background and efficient computation strategies for this property are again examined in the accompanying paper /KSW89/.

In many applications, the term 'identical' is used in a more restrictive fashion for duplicate identification. There, the identity of two structures is decided by inspecting only part of the structure. This occurs especially due to the use of semantic knowledge on the application area or due to optimizing duplicate elimination. In the case of relations, the use of *key attributes* is an example of such a restrictive identity definition - two tuples are assumed to be equal if their key attribute values are equal. In conventional relational databases, this notion is only used for database tables and not for result tables. We use these notions also for result structures and extend the concepts to *key expressions* where identity is defined by the equality of arbitrary expressions computed out of the set or list elements.

As an example for the need of key attributes (for duplicate elimination), we assume a personnel database consisting of personnel data like `Name`, `Age` etc., and an attribute `Picture` containing a 'bitmap' picture of the person. In a personnel relation, duplicate elimination should not compare the binary representation of the pictures to test identity of person records. On the other hand, in a set of fractional numbers we may fix equality by comparing the reduced fractions using a complex HDBL expression. If we use such an expression for determining the identity for duplicate suppression, we talk about *key expressions*. The usual key attributes are included as a special case.

As with the sort methods, the type 'list' needs special attention if key expressions are used for duplicate identifications. Structures with equal key values need not be equal on other parts of the structure. So the question arises, which of the occurring duplicates have to be eliminated. This problem can either be solved by the intuitively acceptable commitment of deleting the duplicate at the higher list position ('the first survives') or controlled by an additional parameter. In the following, we use the first variant.

In the evaluation of database queries, duplicate suppression makes sense in two evaluation phases. The most commonly used duplicate suppression is on the *result of the query*. Duplicate suppression on the result delivers always a result of type unique set (or unique list, if the input was of type list). For this reason, duplicate suppression on the query result can only have expressions over the result structure as key expressions. An example for duplicate suppression on the query result is a unique set of authors derived from the literature table.

The second, more subtle duplicate suppression is on the *input structures* of an SFW expression. This enables key expressions on non-selected data, for example if we want to project the titles out of the literature table only once unless associated with different author lists. This can be expressed by duplicate suppression with the keys `Title` and `Authors` on the input of the corresponding SFW construct (example (22)). It must be noted that the result of a query with duplicate suppression on the input may still contain duplicates.

4.2 Duplicate Suppression on Input

For duplicate suppression we have essentially the same design options for language features as for sorting. The duplicate suppression on input structures is closely related to sort expressions, since the mental model implies the removing of 'input lines' in the same evaluation phase where sorting is done (before result construction). Moreover, in this case legal key expressions are the same as legal sort expressions (without ASC or DESC keywords). We chose a descriptive notation analogous to the ORDER BY expressions :

```
(21.1)  SELECT l.Title FROM l IN Literature
(21.2)  IGNORE DUPLICATES ON l.Title
```

Example (21) shows that variables are bound to the input structures occurring in the FROM-list. This enables the formulation of the (above mentioned) query (22), where duplicate suppression is done with respect to non-selected information :

```
(22.1)  SELECT l.Title FROM l IN Literature
(22.2)  IGNORE DUPLICATES ON l.Title, l.Authors
```

Example (23) shows the use of arithmetic key expressions. The result contains only points with different distances from the origin (0,0).

```
(23.1)  SELECT p FROM p IN Points
(23.2)  IGNORE DUPLICATES ON p.x*p.x + p.y*p.y
```

The syntactical structure of an IGNORE clause is analogous to the order expression (without ASC and DESC). If an ignore and an order expression are combined in one SFW construct, their relative order is relevant due to the 'the first survives' rule for duplicate deletion. The following two SFW queries may deliver different results; in the second variant (25) the author with the 'lexicographic smallest' first name survives, whereas in the first

variant (24) this depends on the order of the input lists.

```
(24.1)  SELECT a                    (25.1)  SELECT a
(24.2)  FROM l in Literature,       (25.2)  FROM l in Literature,
(24.3)       a in l.Authors         (25.3)       a in l.Authors
(24.4)  IGNORE DUPLICATES           (25.4)  ORDER BY a
(24.5)       ON a.Surname           (25.5)  IGNORE DUPLICATES
(24.6)  ORDER BY a                  (25.6)       ON a.Surname
```

4.3 Duplicate Suppression on Result

Now let us return to duplicate suppression in the query result. In the *mental evaluation model* this suppression has to be performed after the result construction phase. To guarantee the uniqueness property on the result structure, only expressions over result table elements are legal key expressions. Following the mental evaluation model, a duplicate suppression on the result structure is always performed *after* the processing of an ORDER or IGNORE clause.

Proposals /CAE76,PA86,PT86/ for relational as well as for extended NF^2 database languages describe several syntactic alternatives. In the language HDBL, the following three alternatives are candidates for an orthogonal language extension:

1. The *functional approach* /PA86,PT86,RKB87/ is similar to the notation of the sort function in example (16). Without keys it may be notated like 'make_unique(Literature)', with key parameters it resembles the functional order operator:

   ```
   (26.1)  UNIQUE x IN ( SELECT [l.Title, a]
   (26.2)                FROM l IN Literature,
   (26.3)                     a IN l.Authors)
   (26.4)    ON x.Title
   ```

   The scope of the variables used in key expressions is bound to the result structure.
2. The *descriptive approach* is similar to the ignore clause described above. Legal key expressions are exclusively built on the basis of a system generated variable RESULT bound to the result structure.

   ```
   (27.1)  SELECT [l.Title, a]
   (27.2)  FROM l IN Literature, a IN l.Authors
   (27.3)  UNIQUE ON RESULT.Title
   ```

   The case of a desired unique result table without key restrictions is formulated by

   ```
   (27.3')   UNIQUE ON RESULT
   ```

3. The *SELECT clause modification approach* is derived from the SQL proposal and the use of key constraints in data definition languages. The SELECT clause defines the structure of the result, and duplicate suppression on the result may be formulated therein. The uniqueness of the result without key restrictions is expressed by the key word UNIQUE which directly follows SELECT, whereas key attributes are marked by KEY. The examples (28) and (29) show the use of both constructs.

   ```
   (28.1)  SELECT UNIQUE l.Title
   (28.2)  FROM l IN Literature

   (29.1)  SELECT [ KEY l.Title, l.Authors ]
   (29.2)  FROM l IN Literature
   ```

   It must be noted that in this approach only first level attributes are usable as key expressions. On the other hand, it may be orthogonal to a DDL extension introducing keys as integrity constraints into the schema definition.

The third alternative has a relevant lack on expressive power compared with the two other variants, so the final decision was between the functional and the descriptive approach. We opted for the second variant for the same reasons as with sorting. This decision also reduces the amount of additional concepts and language features.

4.4 Syntax and Semantics

The duplicate suppression on *input structures* is analogous to the order-by clause:

```
<SFW> :: SELECT <expr> FROM <from-list> [WHERE <bool-expr>]
         [IGNORE DUPLICATES ON <expr> {"," <expr>}*]
```

The mental evaluation model for duplicate suppression on input structures is very similar to the one for sort expression evaluation:

1. see order_by evaluation.
2. see order_by evaluation.
3. Computation of key expression values for the input lines.
4. If two lines have equal key expression values, one of them is removed. If the input structure is of type list, this removing has to obey the relative position stability.
5. Construction of result (see order_by evaluation).

In contrast to the IGNORE clause evaluation, duplicate suppression on the *result structure* allows only expressions over the variable RESULT (<result-expr>) to appear as key expressions.

```
<SFW> :: SELECT <expr> FROM <from-list> [WHERE <bool-expr>]
         [UNIQUE ON <result-expr> {"," <result-expr>}*]
```

One aim of the distinction of both duplicate suppression alternatives is that for duplicate suppression on the result structure we can syntactically derive the uniqueness of the result. To guarantee this property, we have to slightly modify our evaluation model.

1. see order_by evaluation.
2. see order_by evaluation.
3. see order_by evaluation.
4. Perform duplicate elimination for the result structure.
5. Computation of key expression values for the result elements.
6. If two lines have equal key expression values, one of them is removed.

The fourth step guarantees that the result itself is free of duplicates, and this property is not affected by the following steps. The duplicate suppression done wrt the key expressions alone is not sufficient because of the existence of nondeterministic expressions in the HDBL language.

If step 4 were omitted, a key expression with a nondeterministic function (for example conversion of a set into a list) may yield in step 5 different key expression values for identical result elements, and therefore uniqueness cannot be guaranteed.

5. The Grouping Operator

5.1 Grouping in the eNF² Data Model

In contrast to *pure* sorting or duplicate elimination, the 'grouping' or 'nesting' known from NF²-oriented languages *changes the structure of the result*. The fundamental idea is to assemble part of the result into a new created set (or list) such that all elements meet the same grouping condition. Because of this fundamental effect of changing the result structure, we decided to choose a functionally oriented language feature instead of a descriptive one (like the SQL proposal).

There are two alternatives for the structure of the result of a grouping operation. The first one is that the result structure contains only the constructed groups /PA86,PT86/. The new result structure is determined by the input structure, for example a 'set of input type' becomes a 'set of set of input type' after grouping. The disadvantage of this approach is the loss of the explicit grouping criterion information.

In the second alternative, the result is a binary relation between the groups and the group criterion. In terms of extended NF² structures, this structure is a set or list of tuples containing the groups and the grouping criterion values as field values. This structure implies the choice of attribute names as additional parameters of the group operator if we want to avoid system generated attribute names.

The result structure of the first variant can be computed from the result of the second variant by a simple projection on the group field of the result tuples. So we decided to choose the second variant because of its greater expressiveness. This choice implies that our grouping operator needs the following parameters :

1. The input structure, a set or list.
2. The grouping criterion (an expression with variables bound to the elements of the input structure).
3. The attribute identifiers for the resulting groups and the grouping values.
4. The structure description for the group elements (also an expression with variables bound to the elements of the input structure).
5. An (optional) predicate on the input structure (WHERE clause of the SFW construct).

Moreover, we decided to integrate the projection and selection operations into the group function to allow compact notations and to simplify query optimization. The following complex example shows the syntactical constructs chosen for the group function and the parameters:

```
(30.1)  GROUP [l.Title,l.Keywords]
(30.2)  INTO Publications
(30.3)  BY Author : a
(30.4)  FROM l IN Literature, a IN l.Authors
(30.5)  WHERE COUNT(l.Authors) < 4
```

The HDBL statement (30) delivers for each author all his (her) relevant publications (where publications are assumed to be irrelevant for an author if authored by four or more authors).

The result structure of group expression (30) is

```
(31.1)  LIST(TUPLE(Publications: LIST(TUPLE(
(31.4)                                  Title : ...,
(31.5)                                  Keywords : ...)),
(31.6)             Author: TUPLE(Surname : ..., ... ) ))
```

The list types (instead of set types) follow from the list type structure Authors in the FROM list.

5.2 Syntax and Semantics

In more detail, the syntax of a group expression is fixed as follows:

```
<gexpr> ::=  GROUP <expr> INTO <id> BY <id> ":" <expr>
             FROM <from-list> [ WHERE <bool-expr> ]
```

The 'from-list' is built analogously to the SFW construct. To make the reader more familiar with this notation, we present a few further examples. As a first one, we group the complete literature by the number of authors :

```
(32.1)  GROUP l INTO LiteratureSet
(32.2)   BY NrAuthors : COUNT(l.Authors)
(32.3)   FROM l IN Literature
```

The second example (33) groups the literature titles by the set of their keywords.

```
(33.1)  GROUP l.Title INTO Titles
(33.2)  BY Keywords : l.Keywords
(33.3)  FROM l IN Literature
```

Please note that the type of the result of (33) is

```
(34)  SET(TUPLE( Titles:SET(TEXT), Keywords:SET(TEXT)))
```

As a last example, we group our points table by the distance to the origin (0,0) (assuming a function SQRT for computing the square root).

```
(35.1)  GROUP p INTO PointsOfEqualDistance
(35.2)   BY Distance : SQRT(p.x*p.x + p.y*p.y)
(35.3)   FROM p IN Points
```

The precise semantics of a grouping operation can readily be defined by an equivalent HDBL ex-

pression, using already defined language features. A group expression

```
(36.1)  GROUP elementexpr INTO groupname
(36.2)   BY criterionname : criterionexpr
(36.3)   FROM fromlist WHERE predicate
```

is equivalent to the following HDBL expression, using functional composition and duplicate suppression.

```
(37.1)  SELECT [ groupname :
(37.2)             (SELECT elementexpr FROM fromlist
(37.3)              WHERE predicate AND x = criterionexpr),
(37.4)           criterionname : x                          ]
(37.5)  FROM x IN (SELECT criterionexpr FROM fromlist
(37.6)             WHERE predicate UNIQUE ON RESULT)
```

In this expression, the variable 'x' has as values all occurring group criterion values (in consideration of the WHERE clause). This equivalent HDBL expression fixes at the same time the result type of a grouping expression with respect to the from list. It is clear that an efficient implementation does not follow this semantics commitment.

6. Integration into Query Evaluation

6.1 The Sort Manager

The language features introduced in this paper are being implemented using a system component called *sort manager*. The sort manager offers at its call interface a sort function for arbitrary eNF² collections and allows for parameters controlling duplicate suppression and grouping during the sort process. For a discussion of appropriate sort algorithms for these functions see /KSW89/.

In detail, the parameters of a sort manager call contain the following data :

- The collection, an ordered or unordered relation, to be manipulated. The type of this collection is always set or list of tuples. The collection is an intermediate internal result structure, not a persistent database object.
- The order criteria. Allowed order criteria are top level attributes of the collection elements together with an ascending / descending flag.
- The duplicate elimination flag. If this flag is set, equality of the sort criteria values leads to elimination of one of the compared tuples.
- The grouping information. This parameter controls the grouping while performing duplicate elimination. It contains the (top level) group attribute, which must be of type set or list and must not be contained in the order criteria list. If a grouping attribute is defined, each time a duplicate elimination is performed the surviving tuple gets the union of both old grouping attribute values as new grouping attribute value.

The parameter restrictions give a less powerful functionality of the sort manager than introduced for the HDBL language extensions described in chapters 3, 4 and 5, but enable a straightforward implementation of the sort manager. The only critical point of the implementation are set-valued order criteria. In this case, the sort algorithm is recursively called to compute the order according to the 'cardinality first, then minimum' rule. The strategy to implement the full proposals of chapter 3 is described in the following subchapters.

6.2 Integration Strategy

We assume that duplicate elimination is done via sorting (for a discussion of other methods and a justification of this choice see /KSW89/). Later on, we will reduce grouping also to a slightly modified duplicate elimination (see 6.4). Due to these reasons, we concentrate mainly on the evaluation of sort expressions by stating that the used concepts carry over to duplicate suppression and grouping.

Of course, the mental model for sorting with 'computing of the cross product' is not an acceptable implementation idea. Instead of the complete cross product, only the result objects have to be computed along with additional attributes containing the information needed for key expressions or sort / grouping criteria. For example, consider the following expression :

```
(38.1)  SELECT l.Title FROM l IN Literature
(38.2)  ORDER BY COUNT(l.Authors)
```

For evaluation, we can compute the result of the extended query below (39), sort the intermediate result and as a last step remove the artificially added data needed for the sorting. Note that the sorted objects are considerably smaller than full lines of the input structure would be.

```
(39.1)  SELECT [id1 : l.Title, id2 : COUNT(l.Authors)]
(39.2)  FROM l IN Literature
```

This method is independent of the evaluation strategy implemented in a DBMS. It reduces the sort criteria internally to upper level attributes of a table and avoids multiple computations of sort expressions. For duplicate elimination, the result structure expansion enables the handling of both the ignore and the unique clause in an equal fashion.

Conceptually, we divide the query evaluation into the following phases:

1. Query analysis. Creation of internal query evaluation plan and catalog structure.
2. Optimization of the internal query representation.
3. Data retrieval out of the database (evaluation of retrieval part of the internal query representation).
4. Result modification phase (sorting etc).
5. Result processing (browsing, transfer to application software, inserting into database table etc.)

The implementation usually mixes phases 3 and 4 for efficiency reasons; for example, substructures are sorted directly after having been retrieved.

Here we concentrate on the added language features only. We subsume sorting, duplicate suppression and grouping under the notion of *result modification operations*. For generality, we talk about result modification expressions, which mean expressions modifying the result of a previous evaluation step (but not its internal result structure). Due to the definition in section 5.2, these operations are also sufficient to implement grouping. An arbitrary modification expression is composed using the following primitives :

1. **SORT:** This basic modification function has as parameters (among others) a reference to the input structure for the operation (which may be a substructure of a complex NF^2 object). In case of the SORT operator, a list of sort criteria is another parameter consisting of field identifiers with additional declarations like descending or ascending order. SORT also provides a facility for eliminating duplicates on the fly.
2. **Composition:** Modification expressions can be concatenated to sequences (for example, for the combination of sorting and duplicate elimination).
3. **Control:** As a final feature, we need loops over substructures (for their justification see also detailed discussion for optimization).

6.3 Details of the Evaluation Steps

Now we can take a closer look at the special activities needed for sorting and duplicate suppression in the single phases. The group operator is handled afterwards (see 6.4).

In the *query analysis phase*, the result catalog and the query evaluation plan have to be extended for the needed computed values of the sort / key expressions. The first step is the addition of the sort criteria to the result structure as additional attributes (see above example). The query evaluation plan is extended to retrieve the data into these attributes.

In the *optimization phase*, the original query evaluation plan constructed in the analysis phase is restructured for an efficient evaluation. Here we concentrate on optimizations related to the result modification. The following points can lead to optimized query evaluation plans.

- The use of indexes may influence the need for sorting, for example an index may deliver the elements already in the desired order making a sort operation superfluous.
- The arrangement of query modification expressions can be optimized. An important optimization technique is to avoid sorting of subcomponents which are removed afterwards due to duplicate elimination. An example is a query which delivers for each author his coauthors in alphabetic order.

```
(40.1)  SELECT [Author : a,
(40.2)          Coauthors :
(40.3)           SELECT ca
(40.4)           FROM 1' IN Literature,
(40.5)                ca in 1'.Authors
(40.6)           WHERE a ELEMENT OF
(40.7                        1'.Authors
(40.8)                 AND NOT (a = ca)
(40.9)           ORDER BY ca          ]
(40.10) FROM 1 IN Literature,
(40.11)           a IN 1.Authors
(40.12) UNIQUE ON RESULT.Author
```

  In this example it is better to first eliminate the duplicates and then do the sorting on the coauthors of the tuples which survive the duplicate elimination process.
- With key expressions for duplicate elimination, the relative order is irrelevant for the result. Thus, they can be re-arranged according to the costs of the equality tests, and according to their selectivity on the input structure.

In the *data retrieval phase*, we do not need additional activities because of the query evaluation plan modifications in the first phases.

The *result modification phase* contains the data manipulation algorithms needed for the new language features. The needed sort algorithms and order computations for extended NF^2 structures are discussed in /KSW89/ and are skipped in this paper. As a last step, the artificially added information is projected out of the result.

The result processing is not affected by the new language features.

6.4 Implementation of Grouping

As mentioned above, the grouping operator is defined by the use of duplicate elimination. The following algorithm sketch shows an elegant implementation of a grouping operator using the introduced formalism :

1. As a first step, the group expression is internally replaced by a SFW construct having the same result structure as the group expression itself. The example group expression

```
(41.1)  GROUP 1 INTO LiteratureSet
(41.2)   BY NrAuthors : COUNT(1.Authors)
(41.3)   FROM 1 IN Literature
(41.4)   WHERE NOT COUNT(1.Keywords) = 0
```

   is internally replaced by

```
(42.1)  SELECT [LiteratureSet : { 1 },
(42.2)          NrAuthors: COUNT(1.Authors) ]
(42.3)  FROM 1 IN Literature
(42.4)  WHERE NOT COUNT(1.Keywords) = 0
```

   After evaluation of this query, the attribute `LiteratureSet` of type multiset contains exactly one element (for each result object).

2. Now a modified duplicate elimination is executed for the result of the modified query (42) : Whenever duplicates on 'NrAuthors' are detected, the contents of the 'LiteratureSet' attribute are assembled and the surviving duplicate gets the assembled elements as new 'LiteratureSet' attribute (see chapter 4.1). For list type structures, the position stability of the grouped elements can be guaranteed using internally an artificially added position attribute.

7. Conclusions

In this paper we presented a language extension for the SQL-like NF² database query language HDBL, which offers powerful language features for sorting, duplicate suppression and grouping of extended NF² structures. Due to space limitations, some details had to be left out. They can be found in /SLPW89/.

Future work contains the implementation of this language extension as well as its improvement in practical applications. Interesting aspects for future work are the integration of sorting and duplicate elimination into query plan optimization and the requirements for an appropriate sorting algorithm to be included in the DBMS (for the latter see also /KSW89/).

The presented extensions affect only the DML part of the HDBL database language. Possible DDL part extensions and the enforcement of these constraints have to be investigated in the future.

Besides the proposed HDBL language extensions, the developed methods for duplicate elimination are also planned to be used for efficient evaluation of recursive queries as proposed in /Li88/. Here, special requirements like partially presorted input have also to be taken into consideration.

Acknowledgements

The authors would like to thank the colleagues in the AIM-P project, namely P. Dadam, R. Erbe, U. Herrmann, U. Keßler, K. Küspert, E. Roman and N. Südkamp for fruitful discussions and suggestions

References

AB84 S.Abiteboul, 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.

ALPS86 Andersen, F., Linnemann, V., Pistor, P., Südkamp, N.: Advanced Information Management Prototype - User Manual of the On-Line Interface of the Heidelberg Data Base Language (HDBL). Techn. Note IBM Scientific Center Heidelberg TN86.01

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.

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.

Da88 P. Dadam: Advanced Information Management (AIM): Research in Extended Nested Relations. IEEE Database Engineering, Special Issue on Nested Relations, Vol. 11, No. 3, 1988.

DaK86 P.Dadam, K.Küspert et al: A DBMS Prototype to Support Extended NF² Relations: An Integrated View of Flat Tables and Hierarchies. Proc. ACM SIGMOD 86, Washington D.C., May 1986, pp. 356-367.

ISO86 ISO International Organization for Standardization: Draft International Standard ISO / DIS 9075 : Information Processing Systems - Database Language SQL. 1986.

KF88 S. Khoshafian, D. Frank: Implementation Techniques for Object Oriented Databases. Proc. Advances in Object-Oriented Database Systems (K.R. Dittrich, ed.), LNCS 334, Springer-Verlag Berlin, 1988, pp. 60-79.

KSW89 K. Küspert, G. Saake, L. Wegner: Duplicate Detection and Deletion in the Extended NF² Data Model. Proceedings 3rd Intern. Conference on Foundations of Data Organization and Algorithms (FODO 1989), Paris, June 1989. (also Techn. Rep. IBM Scientific Center Heidelberg TR 88.11.012)

Li88 V. Linnemann: Functional Recursion and Complex Objects. Techn. Rep. IBM Scientific Center Heidelberg TR 88.12.017, Dec. 1988.

LKD88 V. Linnemann, K. Küspert, P. Dadam et al: Design and Implementation of an Extensible Database Management System Supporting User Defined Data Types and Functions. Proc. 14th Int. Conf. on Very Large Data Bases, Los Angeles 1988. (also Techn. Rep. IBM Scientific Center Heidelberg TR 87.12.011

PA86 P.Pistor, F.Andersen: Designing a Generalized NF² Model with an SQL-Type Interface. Proc. VLDB 86, Kyoto, Aug. 1986, pp. 278-288.

PHH83 P.Pistor, B.Hansen, M.Hansen: Eine sequelartige Schnittstelle für das NF² Modell. In J.W.Schmidt (ed.): Sprachen für Datenbanken. Informatik Fachberichte 72, Springer Verlag, Berlin-Heidelberg-New York, 1983. pp. 134-147.

PI87 P. Pistor: The Advanced Information Management Prototype: Architecture and Language Interface Overview. IBM Scientific Center Heidelberg Techn. Rep. TR 87.06.004, June 1987.

PT86 P.Pistor, R.Traunmüller: A Database Language for Sets, Lists, and Tables. Information Systems, Vol. 11(4), 1986, pp. 323-336.

RKB87 M.A. Roth, H.F. Korth, D.S. Batory: SQL/NF: A Query Language for ¬NF Relational Databases. Information Systems Vol. 12, No. 1, 1987, pp. 99-114.

SLPW89 G.Saake, V.Linnemann, P.Pistor, L.Wegner: Sorting, Grouping and Duplicate Elimination in the Advanced Information Management Prototype. IBM Scientific Center Heidelberg Techn. Rep. TR 89.03.008, March 1989.

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