REPRESENTATIONS AND QUERY LANGUAGES OF NESTED RELATIONS

Gy. Kovács (Budapest, Hungary)
Cs. Hajas(Debrecen, Hungary)
I. Quilio(Versailles, France)

Abstract. In real life applications it is often needed to store and handle complex objects. The first normal form assumption of the relational model turned out to be too restrictive for such applications, therefore more advanced data models have been investigated for years, e.g. the nested relational model and object oriented data models.

In this paper we present the concept of nested relations. A value in such a relation may be itself a relation, thus allowing to directly represent structured data. We give an overview of the related query languages, such as relational algebra and calculus, and an SQL-like language called NF2-SQL. Since currently we are working on the relational implementation of a DBMS supporting nested relations, we also show in the paper how nested relations can be represented by traditional (flat) relations.

1. Introduction

The relational data model ([2], [13]) has become the most popular data model implemented in many database management systems. However, the assumption that relations are always in first normal form (attributes may hold only atomic values) restricts the applicability of the model in a large extent, because real life applications (e.g. multimedia, geographical applications) often require to model and store highly structured, complex objects.

Supported by the Copernicus LPDB Project, Project Nr. 93:6638.

Partially supported by the Hungarian National Science Grant (OTKA), Grant Nr. T016524.

In the nested relational model (also called NF^2, Non First Normal Form relational model) a value of a relation itself can be a relation. Subrelations may also contain relations as their values, that is the depth of nesting may be arbitrary. This makes it possible to represent structured data in a natural way and also decreases redundancy. The NF^2 model has been investigated for years, several authors have defined the model and extensions of the relational algebra and calculi for nested relations ([3], [4], [5], [7], [9], [12]). Also some SQL extensions have been proposed ([8], [10]).

Although the NF^2 model provides data structuring facility it still has limited applicability. To overcome the limitations of the NF^2 model more advanced data models have been proposed. In [8] an extended NF^2 model is presented which allows additional data types, such as lists and multisets (bag). Others have been working on incorporating object orientated concepts into database technology (e.g. [1]). Although object orientated models are more general than the nested model, from database management point of view they have several critical points, e.g. lack of updates. In our approach the NF^2 model is considered as an intermediate step to overcome some of the problems.

The rest of this paper is organized as follows. In Chapter 1 we formally define the main concepts of the NF^2 model. Chapter 2 gives an overview of the nested relational algebra and tuple calculus, and also some extended algebraic operations are presented. A nested SQL extension is described in Chapter 4, and in Chapter 5 we show how nested relations can be represented with flat relations. In Chapter 5 a short overview is given and a little discussion on object oriented themes.

2. The nested relational model

In this section we introduce concepts of the nested relational model. Our formalism is a modified combination of the approaches found in [5] and [12].

We assume that there is an infinitely enumerable set U of attribute names. Attributes and relation schemes in the nested model are constructed from this set. An attribute has a name and a scheme, where the name is from U and the scheme is a set of attributes, may be empty. We will have two kinds of attributes, atomic attributes are attributes with empty scheme, and attributes with nonempty scheme are called composite (or structured) attributes. Attributes can be referred to by their names if no ambiguity occurs.

Now we formally define these notions and we introduce the function **names** on attributes and schemes which gives the set of attribute names occuring in a given attribute or scheme, respectively.

Definition 2.1. The *set of attributes* $\mathcal{U}$, the *set of schemes* $\mathcal{S}_0$ and the function **names**: $(\mathcal{U} \cup \mathcal{S}_0) \rightarrow 2^U$ are defined recursively as follows:

(1) $\emptyset \in \mathcal{S}_0$ and $\mathbf{names}(\emptyset) = \emptyset$;

(2) $\forall A \in U :< A, \emptyset > \in \mathcal{U}$ and $\mathbf{names}(< A, \emptyset >) = \{A\}$;

(3) If $X_1, ..., X_n \in \mathcal{U}$ and $\forall i, j = 1, ..., n$ with $i \neq j$, $\mathbf{names}(X_i) \cap \mathbf{names}(X_j) = = \emptyset$, then $\{X_1, ..., X_n\} \in \mathcal{S}_0$ and $\mathbf{names}(\{X_1, ..., X_n\}) = \cup_{i=1}^{n} \mathbf{names}(X_i)$;

(4) If $S \in \mathcal{S}_0$, $A \in U$ - $\mathbf{names}(S)$, then $< A, S > \in \mathcal{U}$ and $\mathbf{names}(< A, S >) = \{A\} \cup \mathbf{names}(S)$;

(5) There are no other elements in $\mathcal{U}$ and $\mathcal{S}_0$.

Since an attribute consists of a name and a scheme component, we define two functions on attributes returning these components. Moreover, the sets of atomic and composite attributes are formally defined.

Definition 2.2. The functions **name**: $\mathcal{U} \rightarrow U$ and **sch**: $\mathcal{U} \rightarrow \mathcal{S}_0$ are defined as follows. Let $X = < A, S > \in \mathcal{U}$ be an attribute. Then $\mathbf{name}(X) = = A \in U$ and $\mathbf{sch}(X) = S \in \mathcal{S}_0$.

Definition 2.3. The *set of atomic attributes* $\mathcal{U}_A$ and the *set of composite attributes* $\mathcal{U}_C$ are the following sets: $\mathcal{U}_A = \{X \in \mathcal{U} \mid \mathbf{sch}(X) = \emptyset\}$ and $\mathcal{U}_C = = \{X \in \mathcal{U} \mid \mathbf{sch}(X) \neq \emptyset\}$.

Consequence 2.1. From Definition 2.3: $\mathcal{U} = \mathcal{U}_A \cup \mathcal{U}_C$ and $\mathcal{U}_A \cap \mathcal{U}_C = \emptyset$.

In the classical (flat) relational model a relation scheme is defined as a nonempty set of attributes, where all the attributes are atomic. The same definition is applied to nested relation schemes, but the attributes in the scheme may be both atomic and composite. Obviously, flat relational schemes are a special case of this definition.

Definition 2.4. A *nested relation scheme* Ω is a nonempty scheme. Let $\mathcal{S}$ denote the *set of nested relation schemes*, or briefly relation schemes. Hence $\mathcal{S} = \mathcal{S}_0 - \{\emptyset\}$.

Note the dualism between relation schemes and schemes of composite attributes. This comes from the nested nature of the model. We will see that a nested relation can be viewed as a value of a composite attribute.

On instance level, a (nested) relation instance over a relation scheme is a finite set of tuples over the scheme, where a tuple over a scheme is a mapping that assigns values to the attributes occuring in the scheme. To atomic attributes atomic values are assigned, and for composite attributes the assigned values are relation instances over the corresponding attribute schemes.

A nested relation consists of a nested relation scheme and a relation instance over this scheme. See formal definition below. We do not distinguish between different types of atomic values, we assume a universal infinitely enumerable set V of atomic values.

Definition 2.5. The set of (*nested*) values $\mathcal{V}$, the set I_X of all *(nested) relation instances over* $X \in \mathcal{S}$ and the set T_X of all *tuples over* $X \in \mathcal{S}$ are the smallest sets that satisfy:

(1) $\mathcal{V} \cup (\cup_{X \in \mathcal{S}} I_X)$,

(2) $I_X = \{R \mid R \subseteq T_X$ and R is finite $\}$;

(3) $T_X = \{t : X \to \mathcal{V} \mid \forall Y \in X \cap \mathcal{U}_A : t(Y) \in V$ and $\forall Y \in X \cap \mathcal{U}_C : t(Y) \in$ $\in I_{sch(Y)}\}$.

Definition 2.6. A *nested relation* or briefly a *relation* R is a pair $R =$ $= (\Omega, \omega)$, where $\Omega \in \mathcal{S}$ is a relation scheme and $\omega \in I_\Omega$ is a relation instance over Ω. We extend the function **sch** introduced earlier to relations. For $R =$ (Ω, ω) : $\mathbf{sch}(R) = \Omega$.

Nested relations can be represented in several ways. For relation schemes we can use the usual linear form. With tabular representation we can represent both the scheme and the value of a relation. These will be illustrated in the next example.

Example 2.1. Let us suppose that a big stock stores the orderings of their clients in a database. They store the names of clients, the article names, ordering dates and the quantities. We show an example how the data can be organized in a nested relation called CLIENT_ORDER. The scheme in our formalism can be defined as:

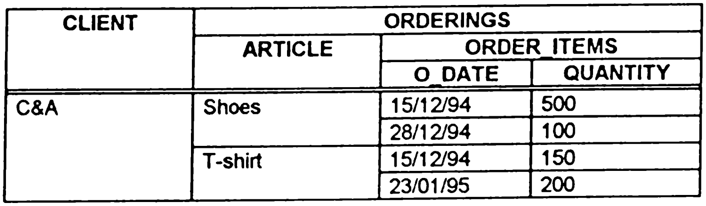
sch(CLIENT_ORDER) = {<CLIENT, $\emptyset$ >, X_1}, where
X_1 = <ORDERINGS, {<ARTICLE, $\emptyset$ >, X_2} >, where
X_2 = <ORDER_ITEMS, {<O_DATE, $\emptyset$ >, <QUANTITY, $\emptyset$ > } >

The linear form of the scheme of the relation:

CLIENT_ORDER(CLIENT, ORDERINGS(ARTICLE, ORDER_ITEMS (O_DATE, QUANTITY)))

The tabular representation of the relation:

Table CLIENT_ORDER:



CLIENT	ORDERINGS		
	ARTICLE	ORDER_ITEMS	
		O_DATE	QUANTITY
C&A	Shoes	15/12/94	500
		28/12/94	100
	T-shirt	15/12/94	150
		23/01/95	200

Fig. 2.1. Tabular representation of a nested relation

3. Nested relational algebra and calculus

The well-known relational algebra and calculi are query languages for the classical relational model. Algebraic operations have been defined in the algebra, while in relational calculi (domain calculus and tuple calculus) queries are expressed in terms of formulas of first order logic. In this chapter we present the extensions of these query systems to query nested relations.

3.1. Nested algebra

Beside the 5 basic operations (*union, difference, Cartesian product, selection* and *projection*) the nested relational algebra contains two additional operations, nest and unnest. We do not present the precise definition of each operation here, only an overview is given and the new operations are defined precisely. The approach is similar to the one in [5] with some augmentation and adaptation to our formalism.

The set operations can be applied to relations with the same scheme and their semantics is as usual. The Cartesian product and projection operations do not require any modification, they are defined in the same way as in the classical relational algebra. In order to be able to express comparisions between composite attributes the set of atomic conditions in the selection operation is augmented with the usual set comparision operators, such as $\subseteq$, $\subset$, $\supseteq$ and $\supset$.

Nest and unnest are new operations in the algebra, they are used for restructuring nested relations. Nest makes a relation more nested, while unnest makes a relation less nested. The content of the relation changes according to the new structure (scheme).

Definition 3.1. *Nest*(ν) operation. Let (Ω, ω) be relation, $X \subseteq \Omega$ and $A \in U$ - **names**(Ω).

$\nu_{X:A}(\Omega, \omega) = (\Omega', \omega')$, where $\Omega' = (\Omega - X) \cup \{< A, X >\}$ and
if $\Omega = X$, then $\omega' = \{\omega\}$,
if $\Omega \neq X$, then

$$\omega' = \{t \in T_{\Omega'} \mid \exists t' \in \omega : t|_{\Omega - X} =$$

$$= t'|_{\Omega - X} \wedge t(A) = \{t"|_X \mid t" \in \omega \wedge t|_{\Omega - X} = t"|_{\Omega - X}\}\}.$$

Definition 3.2. *Unnest*(μ) operation. Let (Ω, ω) be a relation and $X \in \in \Omega \cap \mathcal{U}_C$.

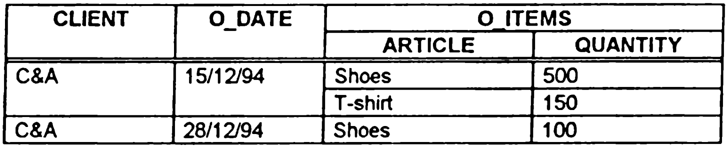
$\mu_X(\Omega, \omega) = (\Omega', \omega')$, where $\Omega' = (\Omega - \{X\}) \cup \mathbf{sch}(X)$ and
$\omega' = \{t \in T_{\Omega'} \mid \exists t' \in \omega : \ t|_{\Omega - \{X\}} = t'|_{\Omega - \{X\}} \wedge t|_{\mathbf{sch}(X)} \in t'(X)\}$.

The operations of the algebra can be applied to one or two relations. Since the result is always a relation, we can build arbitrary algebraic expressions by sequentially applying the operations, as illustrated in Example 3.1.

Example 3.1. Let us take the nested relation from Figure 2.1 and give an algebraic expression which results in a table in wich only the orderings before 31/12/94 are kept and data are grouped by the clients and the ordering dates. The appropriate expression:

$$\nu_{ARTICLE,QUANTITY:O\_ITEMS}(\sigma_{O\_DATE<31/12/94}(\mu_{ORDER\_ITEMS}$$

$$(\mu_{ORDERINGS}(\text{CLIENT\_ORDER}))))$$



CLIENT	O_DATE	O_ITEMS	
		ARTICLE	QUANTITY
C&A	15/12/94	Shoes	500
		T-shirt	150
C&A	28/12/94	Shoes	100

Fig.3.1. Result of the algebraic expression

It is important to note that all the operations of the algebra can only involve attributes at the highest level of the scheme. Queries on lower level can only be expressed by first unnesting the attributes to the highest level, performing the operations and nesting back to the original structure. Unfortunately nest and unnest are not inverse operations, only a nest can always be undone by an unnest.

Proposition 3.1. Let $R = (\Omega, \omega)$ be a relation, $X \subseteq \Omega$ and $A \in U -$ **names**(Ω). Then $\mu_A(\nu_{X:A}(R)) = R$.

There are two reasons why an unnest cannot always be undone by a nest. The first is that if we have two tuples equal on all attributes not being unnested then the corresponding nest will not give back the original two tuples. Instead, it will produce only one tuple with a value (relation) of the unnested attribute which is the union of the two relations being the values of the original two tuples on the unnested attribute. The other reason is that if the attribute being unnested contains empty relations as values, then by definition those tuples will be lost after the unnest.

The latter problem can be solved by using and exact handling of null values ([7], 11]). The first problem can be overcome by uniquely tagging the tuples of the relation. One possibility is that the tag for each tuple is the tuple itself as a composite value. This means that an extra composite attribute is added to the scheme holding these values. We do not give here the precise definition, but we use this tagging operation (τ).

Proposition 3.2. Let $R = (\Omega, \omega)$ be a relation and $X =< A, Y >\in$ $\in \Omega \cap \mathcal{U}_C$. Moreover, let us suppose that for each $t \in \omega : t(X) \neq \emptyset$. Then $R = \pi_\Omega(\nu_{Y:A}(\mu_X(\tau(R))))$.

Proposition 3.2 says that any unnest can be undone by the corresponding nest if the tuples are tagged before unnesting. It is also true that there an algebraic expression can be given to express the tagging operation. The consequence of these is that any lower level operations can be expressed in the defined nested relational algebra.

3.2. Nested tuple calculus

In this section we discuss the nested relational tuple calculus, a logic based approach to query nested relations. In the calculus queries are expressed in terms of logical formulas. Because of lack of space, we give only a short overview.

Formulas are built in the usual way. They contain tuple variables where the variables may have free and bound occurences. Relatively to the classical relational tuple calculus, we have additional atomic formulas to express comparisions between composite attributes. More complex formulas can be obtained by applying the usual logical operators (and, or, not). The nested structure requires the set building formula of the following form to be introduced.

$t(X) = \{s \mid \Psi(s, u, v, ...z)\}$ is a formula if $\Psi(s, u, v, ...)$ is a formula with free tuple variables $s, u, v, ..., z$, $X \in \mathcal{U}_C$, and t is a tuple variable with no free occurrence in Ψ. It expresses that for a given interpretation of tuple variables $t, u, v, ...$, the value assigned to X by t is equal to the set of s satisfying $\Psi(s, u, v, ...)$ with the given $u, v, ...$ interpretations. If there is no such s, then the interpretation: false.

In the nested tuple calculus a query has the following form:

$$\{t \mid \Psi(t)\}.$$

$\Psi(t)$ is a formula with one free tuple variable t. The result of this query is a nested relation containing tuples that satisfy Ψ.

To avoid queries of which result is not computable in finite time, the notion of safety is also extended for formulas of the nested calculus. Roughly speaking, safe formulas are formulas having only variables limited by the content of the database (set of relations) being queried.

Dealing with the flat relational model we know that the relational algebra and the relational tuple calculus have equivalent expressive power. The following theorem states that the same holds for their nested extensions. The proof is omitted in the paper.

Theorem 3.1. *The nested relational algebra and the nested relational tuple calculus are equivalent in expressive power.*

3.3. Extended operations

Although we saw in Section 3.1 that queries involving lower level attributes can be expressed by the operations of the nested algebra, it is still desirable to have operations that can be directly applied on lower level of relations ([3], [12]). Beside convenience, the other reason is that a sequence of unnest and nest is not an efficient way of performing such a query. In this section we introduce some extended operations. Our approach is similar to the approach in [3].

Let us first consider the set operations. For instance the way how union works does not necessarily meets the wishes of the user. If we take the union of two nested relations each having a tuple with the same values on all the atomic attributes, the result will contain both individual tuples. However, the user may want to see only one tuple with the common atomic values and the subrelations in the tuple should be the unions of the corresponding subrelations of the original two tuples. That is the union should be done on each level of nesting. Similar consideration can be made for the difference and intersection operations. With keeping the originally defined set operations, extended set operations are introduced in the algebra with recursive semantics. In the paper we only show the definition of the extended union, the others are defined similarly. An example is shown on Figure 3.1.

Definition 3.3. *Extended union* ($\cup^e$). Let (Ω, ω_1) and (Ω, ω_2) be relations and $A = \Omega \cap \mathcal{U}_A$.

$(\Omega, \omega_1) \cup^e (\Omega, \omega_2) = (\Omega, \omega')$, where

$$\begin{aligned} \omega' = \{t \in T_\Omega \mid\ & (t \in \omega_1 \wedge \forall t_2 \in \omega_2 : t_2|_A \neq t|_A) \\ & \vee (t \in \omega_2 \wedge \forall t_1 \in \omega_1 :\ t_1|_A \neq t|_A) \\ & \vee (\exists t_1 \in \omega_1 \wedge \exists t_2 \in \omega_2 :\ (t|_A = t_1|_A = t_2|_A \\ & \wedge \forall X \in \Omega \cap \mathcal{U}_C :\ t(X) = t_1(X) \cup^e t_2(X)))\}. \end{aligned}$$

a1	b1
	b2
a2	b2
	b3

R1

a2	b3
	b4
a3	b1

R2

a1	b1
	b2
a2	b2
	b3
a2	b3
	b4
a3	b1

R1 $\cup$ R2

a1	b1
	b2
a2	b2
	b3
	b4
a3	b1

R1 $\cup^e$ R2

Fig.3.2. Union and extended union

The other operations (Cartesian product, selection, projection, nest and unnest) can also be extended in such a way that they can be applied to lower level attributes. We cannot give all the definitions here, we take the selection to illustrate this mechanism.

Definition 3.4. Let $S \in \mathcal{S}$ be a relation scheme. L is a *selection list on* S if L is empty, or $L = \{< X_{1F_1}, L_1 >, \ldots, < X_{nF_n}, L_n >\}$, $n \geq 1$, where $X_i \in S \cap \mathcal{U}_C$, F_i is a selection condition on $\mathbf{sch}(X_i)$ and L_i is a selection list on $\mathbf{sch}(X_i)$. $X_i \neq X_j$ if $i \neq j$. If F_i is empty then interpreted as true.

In the above definition a selection condition on a scheme means that the condition involves only attributes occuring in the scheme.

Definition 3.5. *Extended selection* (σ^e). Let (Ω, ω) be a relation, let F be a selection condition on Ω and let L be a selection list on Ω. Then $\sigma^e_{F,L}(\Omega, \omega) = (\Omega, \omega')$, where

if L is empty, then $\omega' = \{t \in \omega \mid F(t)\}$,

if $L = \{< X_{1F_1}, L_1 >, \ldots, < X_{nF_n}, L_n >\}$, $n \geq 1$, then

$$\begin{aligned} \omega' = \{t \in T_\Omega \mid \exists t' \in \omega :\ & (F(t') \wedge t|_Y = t'|_Y \wedge \\ & t(X_i) = \sigma^e_{F_i, L_i}(\mathbf{sch}(X_i), t'(X_i)) \neq \emptyset,\ i = 1, \ldots, n)\}, \end{aligned}$$

where $Y = \Omega - \{X_1, \ldots, X_n\}$.

The rest of the extended operations are similarly defined, some auxiliary notions are introduced, such as projection list for projection, nest list for nest, etc. We note that the extended operations can be expressed in the basic nested algebra.

4. NF2-SQL: An SQL extension for nested relations

SQL is the most widely used query language for relational databases. We have defined an SQL extension, referred to as NF2-SQL, which enables to query nested relations. Recently, we have been working on the compilation of NF2-SQL and in the first implementation only a simplified SQL extension has been developed, similar to the one presented in [10]. In this section we present the most important aspects of our extension. The subsections are organized according to the three main parts of the language, the data definition language (DDL), the query language and the data manipulation language (DML). The data control part (DCL) is not discussed in the paper.

4.1. Data definition

The data definition language (DDL) contains statements to create and drop nested tables (relations) and views and to change the schemes of tables.

Like in SQL, the CREATE TABLE statement has two forms, one creates an empty table with the specified column (attribute) names and data types, and the other creates a table based on a given query and the table will be initialized with the result of the query.

In the former case the keyword TABLE is introduced to describe composite columns, and the own scheme of such a column is specified after the keyword in the usual way. It can also be specified if a column is nullable or not (WITH NULL or NOT NULL). As we will see in Chapter 5, users may choose the way how composite values (subtables, table-values) should be represented. The keywords BY VALUES and BY COLUMN stand for value and column representations, respectively. See next chapter for details about this.

Example 4.1. The following statement creates the nested table of Figure 2.1.

```
CREATE TABLE CLIENT_ORDER
          (CLIENT      CHAR(20),
          ORDERINGS     TABLE BY COLUMN NOT NULL
               (ARTICLE     CHAR(15),
                 O_ITEMS     TABLE BY VALUES
```

(*O_DATE* DATE,
QUANTITY INTEGER)));

A view can be created with the CREATE VIEW statement. After the AS keyword a query must be specified which defines the view. Optionally, the whole (nested) scheme of the view can be given which must be compatible with the result of the query. Nested tables and views can be dropped with the DROP TABLE and DROP VIEW statements. No changes to the corresponding SQL statements are required.

The schemes of nested tables can be changed with the ALTER TABLE statement. We can ADD, DROP and MODIFY columns on arbitrary level of nesting. This is not further detailed.

4.2. Data retrieval

The most essential part of a database language is the query language to retrieve data stored in the database In SQL it consists a single SELECT statement, which is general enough to express all the queries expressable in the relational algebra and calculi. Even more sophisticated queries can be expressed in SQL with aggregate functions (COUNT, SUM, etc.).

The query language part of NF2-SQL, discussed in this section, consists of the extended SELECT statement, and according to the new operations of the nested algebra, it contains two additional statements, namely NEST and UNNEST. First we present the essential aspects of our extension and then we give some example queries.

A query in NF2-SQL has the following form:

<subselect> { < set_operator > <subselect> }
[ORDER BY *order_column* [ASC | DESC] {, *order_column* [ASC | DESC] } , where
<set_operator> ::= [DEEP] { UNION [ALL | DISTINCT] | INTERSECT | EXCEPT }

The subselects may be SFW expressions, NEST statements or UNNEST statements, detailed later, all yielding compatible nested tables. The semantics of the above syntax is that the given set operations are performed on the tables resulted by the subselects and the final result is ordered by the columns (only atomic) specified in the ORDER BY clause. The keyword DEEP before a set operator stands for the recursive interpretation of the operation, see extended operations in Section 3.3. For UNION the ALL keyword prescribes to keep multiple tuples, like in SQL.

SFW expressions consist of the same clauses like in SQL, namely SELECT, FROM and optionally WHERE, GROUP BY and HAVING, and also the semantics is the same. First the Cartesian product of the tables given in the FROM clause is performed, then the tuples satisfying the search condition of the WHERE clause are selected. If there is GROUP BY clause then the remaining tuples are partitioned based on the equality of values in the listed columns. The partitions are further filtered by the search condition of the HAVING clause, if there is. Finally, the expressions given in the SELECT clause are evaluated.

In NF2-SQL some essential extensions for SFW expressions have been introduced. In the FROM clause not only table and view names are allowed, but arbitrary queries can be given to specify tables. This accords to the principle of orthogonality in programming language design (see also [10]). The given queries are evaluated before performing the Cartesian product. We can assign names (iterator variables) for the queries to refer to their tuples.

The SELECT clause of an SFW expression contains expressions. We distinguish between atomic expressions and nested expressions. Atomic expressions are the ones allowed in SQL, that is they are built from constants of atomic type, outermost level atomic column names, arithmetical operators and built in function calls. Obviously, atomic expressions are not enough to express queries on subtables, therefore we allow nested expressions to be nested in the SELECT clause.

Nested expressions are either highest level composite column names or restricted SFW expressions. The restriction applies to the FROM clause of the nested SFW expression. Such a FROM clause may contain at most one (in most cases exactly one) outermost level composite column name (not subsitutable with query) and some other tables specified by name or query. In the evaluation of a nested SFW expression for a tuple, the table-value of the tuple in the given composite column is taken. Column names in the result can be specified like in SQL.

The search condition in the WHERE clause is built from predicates and the logical operators AND, OR and NOT. Predicates are atomic conditions. Comparing to SQL we have introduced the following extensions. For comparision between composite columns we have the CONTAINS and SUBSET OF predicates with their usual meaning. With the IN predicate we can build tuples and examine if they occur in (sub)tables, like ('John', 20, ...) IN (SELECT ...). All the other predicates, e.g. LIKE, EXISTS, etc., are kept without changes.

The aggregate functions COUNT, SUM, AVG, MAX and MIN are special built in functions known from SQL. They can occur in the SELECT clause of SFW expressions and can be applied to expressions not containing aggregation calls. The set of values that correspond to the argument expression for each tuple is taken and the result is a single value, that is the result of such a query is a table with one tuple. In NF2-SQL aggregate funcions can be applied only to atomic expressions, except COUNT which can be applied also to composite columns.

If we specify GROUP BY clause then the set of tuples is partitioned and the aggregate functions are evaluated for the partitions separately giving a value for each partition. That is the result may contain several tuples. We allow only atomic columns in the GROUP BY clause. We can also filter the set of partitions, the selection criteria can be given in the HAVING clause. Similarly to SQL, the HAVING condition may contain aggregate functions.

An important point is that if aggregation is applied in a nested SFW expression then an implicit unnest is performed if there is no GROUP BY, because from user point of view the single tuple of the result of the expression is part of the higher level tuple. If there is GROUP BY then the result is a table-value, therefore no unnest should be done, even if the result contains only one tuple.

The syntaxes of the NEST and UNNEST statements are as follows:

NEST <table>
ON *column_name* {, *column_name*} AS *new_column_name*
UNNEST <table>
ON *column_name* {, *column_name*}

The table can be specified by an arbitrary query in both statements. The columns in NEST must be on the same level of nesting. The semantics of the statements are defined by the corresponding algebraic operations. As can be seen, more than one unnest can be performed with a single UNNEST statement, all the columns must be composite.

Example 4.2. Let us take the CLIENT_ORDER table of Figure 2.1 and the following ARTICLES table:

ART_NAME	PRICE
Shoes	3500
T-shirt	1200
Jeans	2500

Give the articles ordered by C&A, and for each article list the quantities of ordering items where the date of order was before 1995. In NF2-SQL:

```
SELECT ARTICLE, (SELECT QUANTITY
                 FROM ORDER_ITEMS
                 WHERE O_DATE < 01/01/95)
FROM (UNNEST CLIENT_ORDER
          ON ORDERINGS)
     WHERE CLIENT = 'C&A' ;
```

The result of the query:

ARTICLE	ORDER_ITEMS
	QUANTITY
Shoes	500
	100
T-shirt	150

For clients with first letter 'C' give how much they spent on the days when they oredered something. Only those days must be listed when the client spent more then 300000. Let the scheme of the result be the following: (CLIENT, DAY_SPENT(O_DATE, TOTAL)).

```
     SELECT CLIENT, (SELECT O_DATE, SUM( PRICE*QUANTITY)
AS TOTAL
                    FROM ORDERINGS AS O2 , ARTICLES AS A
                    WHERE A.ART_NAME = O2.ARTICLE
                    GROUP BY O_DATE
                    HAVING TOTAL > 300000 ) AS DAY_SPENT
     FROM (UNNEST CLIENT_ORDER
          ON ORDER_ITEMS ) AS O
     WHERE O.CLIENT LIKE 'C% ' ;
```

The result is:

CLIENT	DAY_SPENT	
	O_DATE	TOTAL
C&A	15/12/94	1930000
	28/12/94	350000

4.3. Data manipulation

The DML of NF2-SQL consists of the INSERT, DELETE and UPDATE statemens for adding new tuples to a nested table, removing tuples from a table and modifying tuples in a table, respectively. In SQL all these statements always refer to database tables. However, in nested tables, where tables may contain subtables, it should be possible to perform these operations on subtables. Since any update operation on a subtable means a modification of the tuple in which the subtable occurs, the operation can be embedded in an appropriate UPDATE statement on one level higher (see also [10]). This will be the mechanism to express database updates on subtables occuring on lower level of nesting.

With the INSERT statement we can add new constant tuples to a table, or the tuples to be inserted can be specified by a query. We do not necessarily must assign values to all attributes, but in those cases we have to give the ones we want to. When assigning values to composite columns, table constants, table names or queries can be given.

The deletion of tuples from a nested table is based on selection criteria which must be given in the WHERE clause of the DELETE statement. The search conditions may be the same as in the SELECT statement. The tuples satisfying the condition are deleted from the table.

To modify tuples in a table UPDATE statement is needed. The SET clause of the statement specifies what new values are assigned to which columns. For composite columns the new values can be any expressions yielding tables of the appropriate type. Remember the role of UPDATE to carry any DML statements to be applied to subtables. This means that on the right side of the assignments in the SET clause also DML statements may occur. During UPDATE only tuples are modified that satisfy the search condition given in the WHERE clause of the UPDATE statement.

Example 4.3. Delete those ordering items of C&A where the article is T-shirt and the date is 15/12/94.

```
UPDATE CLIENT_ORDER
SET ORDERINGS = (UPDATE ORDERINGS
                 SET ORDER_ITEMS = (DELETE FROM ORDER_ITEMS
                                    WHERE O_DATE=15/12/94)
                 WHERE ARTICLE = 'T-shirt')
WHERE CLIENT = 'C&A';
```

5. Relational representation

As we said before, we have been implementing a DBMS supporting nested relations. We build our system on relational DBMSs, such as Ingres and Oracle. In this approach nested tables are represented with flat relational tables and NF2-SQL statements are translated to SQL statements manipulating those representing tables. In this chapter we shortly present how we represent nested relations with flat tables.

One possibity is that table-values occuring in nested tables are stored in separate flat tables and the tuples to which the subtables belong to only contain references (table names) to the corresponding flat tables. If the subtables themselves contain table-values then the same principle can be applied recursively. This way of representation is referred to as the *value representation.*

Since subtables being in the same composite column have the same scheme, they can be stored in one flat table. However, in such representation we must be able to determine that which tuples of the flat table belong to the same table-values and to which higher level tuples those table-values belong. To solve these problems we add a new column (DOWN) to the flat table representing the (outermost) nested relation and we put unique tuple identifiers into this column. The flat table storing the tuples of subtables occuring in a composite column is also augmented with an additional column (UP) where references (tuple identifiers) to the higher level tuples are placed. Notice that doing this we have solved the whole problem, because with an appropriate selection on the flat table we can get the tuples that belong to a given higher level tuple. This representation can be applied recursively on each level of nesting and is referred to as the *column representation.*

Since the described representations are related to composite columns and not tables, it is possible to mix the value and column representations when representing a nested table. On the other hand for perfomance reasons it is good to keep both representations. The choice is passed to the user by allowing the TABLE BY VALUES and TABLE BY COLUMN options in the CREATE TABLE statement of NF2-SQL.

Obviously, not only the contents of nested relations must be represented, but also some meta data, e.g. the schemes of nested relations. We do not further discuss this part, we only note that all the meta data are stored in flat tables and are maintained by the system via SQL.

6. Conclusions, towards object-oriented databases

In this paper we have given an overview of the nested relational model, an extension of the relational data model where the first normal form assumption is omitted. The nested relational algebra and calculus have been discussed and an SQL extension, called NF2-SQL, have been presented. Finally, we have shown how nested tables can be represented with flat relational tables.

Although the nested model provides structuring facility, it is not satisfactory for more advanced applications, e.g. multimedia and geographical applications. More general and powerful models can be obtained by integrating database models with object oriented concepts. Working on such models have become very popular nowadays (e.g. [1], [6]).

In [1] a database model called ODMG is provided as a standard proposal for object oriented databases. An object database consists of collections of objects. All the well known object oriented concepts (classes, methods, inheritence, object identity, polymorphism, etc.) are parts of the model. Also they make a clear distinction between attributes and relationships. An object definition language (ODL) is given to create types and objetcs. They also propose an object query language (OQL) for querying collections with an SQL-like syntax. Although the proposed model fits into the object oriented paradigm, from database management point of view it has several critical points (see [6]), e.g. there are no database update operations, views are not supported, invocation of methods in queries are not controlled, etc. We think that a good implementation is quite far because of the technical problems.

The ODMG model can be considered as a generalization of the nested relational model. In our view the nested model is considered as an intermediate step towards object oriented databases. We believe that some of the problems, e.g. lack of updates, might be overcome by investigating how the solution for the nested model could be generalized. Our future work will be focussed on such investigations.

References

[1] *The object database standard: ODMG-93,* ed. R.G.G. Cattell, Morgan Kaufmann Publisher, San Francisco, 1994.

[2] **Codd E.F.,** A relational model of data for large shared data banks, *Commun. ACM,* **13** (6) (1970), 377-387.

[3] **Colby L.S.,** A recursive algebra for nested relations, *Inf. Systems,* **15** (5) (1990), 567-582.

[4] **Garnett L. and Tansel A.U.,** Equivalence of the relational algebra and calculus for nested relations, *Computers Math. Applic.,* **23** (10) (1992), 3-25.

[5] **Gyssens M., Paredaens J. and van Gucht D.,** A uniform approach towards handling atomic and structured information in the nested relational database model, *J. ACM,* **36** (4) (1989), 790-825.

[6] **Kim W.,** Observations on the ODMG-93 proposal for and object-orientated database language, *SIGMOD RECORD,* **23** (1) (1994).

[7] **Levene M. and Loizou G.**, Semantics for null extended nested relations, *ACM Trans. Datab. Syst.*, **18** (3), (1993), 414-459.

[8] **Pistor P. and Andersen F.**, Designing a generalized NF2 model with an SQL-type language interface, *Proc. 12th VLDB, Kyoto, Japan, 1986,* 278-285.

[9] **Paredaens J., de Bra P., Gyssens M. and van Gucht D.**, *The structure of the relational database model,* Springer, 1989.

[10] **Roth M.A., Korth H.F. and Batory D.S.**, SQL/NF: A query language for ¬1NF relational databases, *Inf. Systems,* **12** (1) (1987), 99-114.

[11] **Roth M.A., Korth H.F. and Silberschatz A.**, Null values in nested relational databases, *Acta Inf.*, **26** (1989), 615-642.

[12] **Schek H.J. and Scholl M.H.**, The relational model with relational-valued attributes, *Inf. Systems,* **11** (2), (1986), 137-147.

[13]**Ullman J.D.**, *Principles of database and knowledge-base systems, Vol.1.*, Computer Science Press, Rockville, Md., 1988.

Gy. Kovács
Laboratory of Informatics
Computer and Automation Institute
Hungarian Academy of Sciences
Lágymányosi u. 11.
H-1111 Budapest, Hungary
kgy@ilab.sztaki.hu

Cs. Hajas
Inst. of Mathematics and Informatics
Kossuth Lajos University
H-4010 Debrecen, Pf. 12.
hajas@math.klte.hu

I. Quilio
PRISM Laboratory
University of Versailles
45, avenue des Etats-Unis
78035 Versailles Cedex, France
Isabelle.Quilio@prism.uvsq.fr