A Recursive Algebra and Query Optimization for Nested Relations

Latha S. Colby

Department of Computer Science
Indiana University, Bloomington, IN, 47405

Abstract

The nested relational model provides a better way to represent complex objects than the (flat) relational model, by allowing relations to have relation-valued attributes. A recursive algebra for nested relations that allows tuples at all levels of nesting in a nested relation to be accessed and modified without any special navigational operators and without having to flatten the nested relation has been developed. In this algebra, the operators of the nested relational algebra are extended with recursive definitions so that they can be applied not only to relations but also to subrelations of a relation. In this paper, we show that queries are more efficient and succinct when expressed in the recursive algebra than in languages that require restructuring in order to access subrelations of relations. We also show that most of the query optimization techniques that have been developed for the relational algebra can be easily extended for the recursive algebra and that queries are more easily optimizable when expressed in the recursive algebra than when they are expressed in languages like the non-recursive algebra.

1. Introduction

The traditional relational model introduced by Codd [4] in 1970 requires that relations be in first normal form (1NF), *i.e.*, all values in a relation must be atomic or non-decomposable. Although this model is sufficient for representing objects that have simple domains, the first normal form assumption makes it difficult to model complex objects. Database applications in the areas of office automation, textual data and engineering designs, involve complex objects and the relational model is unsuitable for such applications. Normalization in the relational model causes a lot of fragmentation in the representation of objects. Information about objects and their relationships is distributed over several different flat tables. This in turn causes queries to be slow and complicated since excessive joins have to be performed among the various relations in the database.

The nested relational model, also sometimes called NF^2 (non-first normal form), is a generalization of the traditional relational model without the first normal form assumption. Attributes of a nested relation can have non-atomic values. Their values can be relations which can hence be viewed as subrelations of the nested relation. The nested relational model allows users to view the database in a way that is closer to their concept of the real world since complex objects can be represented as a whole in a single nested relation instead of being distributed over several different flat relations. This model was first proposed by Makinouchi [13] who suggested that the 1NF assumption be relaxed since it was too restrictive. Jaeschke and Schek [11] proposed a generalization of the relational model by allowing relations to have non-atomic or set-valued attributes. Thomas and Fischer [25] generalized the model of Jaeschke and Schek by allowing relations to have relation-valued attributes, and since then a number of researchers [1,10,14,16,17,18,19,22,26,27] have extended the relational database theory to nested relations. Several groups [2,6,7,9,15,24] are attempting to implement the nested relational model, either directly or on top of an existing DBMS.

The tables in Figure 1 are an example of a database for a stock-brokerage firm represented in the (flat) relational model. Figure 2 shows the same information in a nested relational schema. In Figure 2, the relation CLIENTS has the relation-valued attribute INVESTMENTS, which in turn has SHARES as a relation-valued attribute. The relation CLIENTS has only two levels of nesting. In general, relations can be nested to any arbitrary but finite depth, *i.e.*, relations can have relation-valued attributes, which can have relation-valued attributes and so on.

CLIENTS

NAME	COMPANY	PURCHASE PRICE	DATE	NO.
John Smith	XEROX	64.50	02/10/83	100
John Smith	XEROX	92.50	08/10/87	500
John Smith	IBM	89.75	06/20/83	200
John Smith	IBM	96.50	11/10/84	100
Jill Brody	EXXON	35.00	01/30/81	100
Jill Brody	EXXON	64.50	01/30/82	100
Jill Brody	EXXON	59.50	02/10/83	200
Jill Brody	FORD	35.50	02/10/83	200
Jill Brody	SEARS	35.75	12/25/87	100

CLIENT INFO.

NAME	ADDRESS
John Smith	311 East 2nd. St. Bloomington, IN 47401
Jill Brody	41 North Main St. Oberlin, OH 44074

STOCK DATA

COMPANY	CURRENT PRICE	LAST DIVIDEND
XEROX	52.25	0.44
IBM	97.50	1.25
EXXON	90.00	0.82
FORD	41.75	0.20
SEARS	77.50	0.34

EXCHANGE DATA

COMPANY	EXCHANGES
XEROX	NEW YORK
IBM	NEW YORK
IBM	LONDON
IBM	HONG KONG
IBM	TOKYO
EXXON	NEW YORK
EXXON	LONDON
EXXON	TOKYO
FORD	NEW YORK
SEARS	NEW YORK

Figure 1. An example of a flat relational database.

CLIENTS

NAME	ADDRESS	INVESTMENTS			
		COMPANY	SHARES		
			PURCHASE PRICE	DATE	NO.
John Smith	311 East 2nd. St. Bloomington, IN 47401	XEROX	64.50	02/10/83	100
			92.50	08/10/87	500
		IBM	89.75	06/20/83	200
			96.50	11/10/84	100
Jill Brody	41 North Main St. Oberlin, OH 44074	EXXON	35.00	01/30/81	100
			64.50	01/30/82	100
			59.50	02/10/83	200
		FORD	35.50	02/10/83	200
		SEARS	35.75	12/25/87	100

STOCK DATA

COMPANY	CURRENT PRICE	EXCHANGES TRADED	LAST DIVIDEND
		EXCHANGES	
XEROX	52.25	NEW YORK	0.44
IBM	97.50	NEW YORK LONDON HONG KONG TOKYO	1.25
EXXON	90.00	NEW YORK LONDON TOKYO	0.82
FORD	41.75	NEW YORK	0.20
SEARS	77.50	NEW YORK	0.34

Figure 2. An example of a nested relational database.

Algebra and calculus based query languages have been proposed for the nested relational model by Roth, Korth and Silberschatz [20], Thomas and Fischer [25], Schek and Scholl [22], and several others [8,10,12,14,16,18]. Some of them [10,14,20,25] are merely simple extensions of the query languages that were developed for the relational model. The nested relational algebra of Thomas and Fischer [25] is one such example where operators like selection and projection have similar definitions for both the relational and the nested relational model, *i.e.*, they can only operate at the outermost level of a relation even if it is a nested relation with relations embedded at several different levels. If a query on a nested relation involves attributes that are nested deep inside the structure of the relation, the relation has to be *flattened* until the attributes of interest are at the outermost level. After the necessary operations are performed on the flattened relation, the result may have to be transformed back into the structure of the original nested relation. Restructuring a nested relation to a relation that has one more or one less level of nesting is done in the nested relational algebra using *nest* and *unnest* operators. So, although the nested relational model provides a better way of modeling complex objects, query languages like the nested relational algebra of Thomas and Fischer cause the structures representing these objects to be restructured while being queried upon; thus defeating the main objective of letting the user operate in an environment that models the real world as closely as possible.

Deshpande and Larson [8] have proposed an algebra for nested relations in which queries can be formulated in a way such that relations do not have to be flattened in order to access and manipulate data at interior levels of relations. This is done by means of an extended selection and a subrelation constructor. The condition for the selection can contain any algebraic expression over the relation-valued attributes of the relation. The subrelation constructor allows relations to be accessed and modified at all levels by creating new subrelations while traversing the different levels of a relation. Although this construct provides navigational ability, it is not very convenient for users to have to introduce new relations and names for these new relations within the query.

Schek and Scholl [22] introduced an algebra for nested relations that enables subrelations at all levels of a relation to be accessed without nesting and unnesting. However, their algebra has several limitations. Firstly, only one operator, *viz.* π, is used as a *navigator*. For instance, if the user wants to unnest a subrelation nested deep inside a relation, he would have to write the query in terms of a series of projections and an unnest. Secondly, renaming is necessary in a number of situations, which is an inconvenience for the user. Finally, the algebra is rather complicated.

Roth, Korth and Batory [18] have proposed an extension to SQL for nested relations which allows nested application of queries so that relations do not have to be restructured. The VERSO algebra, proposed by Abiteboul and Bidoit [1] is probably closest to the algebra presented in this paper although not all of the operators in their algebra can access subrelations without restructuring. Also, their algebra is defined only on a subclass of nested relations called Verso relations, which are nested relations in Partitioned Normal Form [19].

In this paper, we present an algebra for nested relations in which the operators of the classical nested relational algebra [25] are extended so that they can be applied not only at the outermost level of a nested relation, but also to subrelations at all levels in a nested relation. This algebra, which is equivalent in expressive power to the nested relational algebra, enables queries to be expressed more naturally and succinctly with fewer restructuring operations. We then consider query optimization principles for this recursive algebra and show that most of them are a natural extension of the optimization principles that have been developed for the relational algebra. Although query optimization for the relational model is an area in which extensive research has been done, the same is not true for the nested relational model. Some results have been shown in the context of efficiently processing relational select-project-join queries by equivalent queries in the NF^2 model by Scholl [23] and in the Verso model by Bidoit [3]. In the relational algebra, query optimization is based primarily on the commutativity and associativity properties of operators. Although most of these properties also hold in the case of the operators of the nested relational algebra, queries are often difficult to optimize since most expressions in this algebra involve the nest and unnest operators which do not commute with each other and with several other operators [25]. In the recursive algebra, nest and unnest are used only when restructuring is really necessary, *i.e.*, they are not needed for accessing the interior levels in a nested relation and so most queries can be formulated by expressions that do not involve these two operators, allowing queries to be optimized more efficiently than in other query languages.

2. The Nested Relational Model

We first give a few definitions for the nested relational model and then introduce the recursive algebra. In this paper, we give definitions for only selection, projection and join. A full set of definitions for all the operators can be found in [5]. We will use the (non-recursive) algebra of Thomas and Fischer [25] as a basis for comparing the advantages of the recursive algebra over query languages whose operators can be applied only at the

outermost level of a nested relation. The definitions for selection, projection and join in the non-recursive algebra are very similar to the corresponding definitions in the relational algebra (except for minor extensions like allowing set comparisons in the condition for selection etc.) and are hence omitted here.

Let $\mathcal{A}$ be the universal set of attribute names and relation scheme names. A relation scheme of a nested relation is of the form $R(S)$ where $R \in \mathcal{A}$ is the relation scheme name and S is a list of the form $(A_1, A_2, ..., A_m)$ where each A_i is either an atomic attribute or a relation scheme of a subrelation. If A_i is a relation scheme of the form $R_i(S_i)$, then R_i, the name of the scheme, is called a relation-valued attribute of R.

For each atomic attribute $A_i \in \mathcal{A}$ let D_i be the corresponding domain of values. An instance r of a relation scheme $R(S)$, where $S = (A_1, A_2, ..., A_m)$, is a set of ordered m-tuples of the form $(a_1, a_2, ..., a_m)$ such that (i) if A_i is an atomic attribute, then $a_i \in D_i$. (ii) if A_i is a relation scheme, then a_i is an instance of A_i. An instance of a relation scheme is also referred to as a (nested) relation. If $R(S)$ is a relation scheme, then $Attr(R)$ is the set of all (atomic and relation-valued) attribute names in S. $RAttr(R)$ is the set of all relation-valued attributes in S and $FAttr(R)$ is the set of all atomic attributes in S. Henceforth, when we refer to a relation scheme, we will refer to it by its name alone, *e.g.*, R instead of $R(S)$.

Let r be an instance of R and let $t \in r$ (a tuple in relation r). If $A \in Attr(R)$ then $t[A]$ is the value of t in the column corresponding to A. If $B \subseteq Attr(R)$ then $t[B] = t[A_1]t[A_2]...t[A_n]$ where $A_i \in B$ $(1 \leq i \leq n)$.

In the example nested relation shown in Figure 3, $R(A, B(C, D(E)), F(G, H))$ is the scheme of the relation, $Attr(R) = \{A, B, F\}$, $FAttr(R) = \{A\}$, and $RAttr(R) = \{B, F\}$.

2.1 The Recursive Algebra

The main motivation for the recursive algebra is to provide a query language which can extract and manipulate data at all levels of a nested relation without having to restructure the relation in order to access data that is nested at various levels in the relation. The operators in this algebra are all recursively defined so that each operator can be applied to subrelations at all levels thus eliminating the need for a special operator to serve as a "navigator".

Selection σ

The selection operator introduced here allows selection conditions to be specified not only on the entire relation, but also on subrelations at all levels of nesting in a relation.

A	B		F	
	C	D / E	G	H
a_1	c_1	e_1, e_2	g_1	h_1
			g_1	h_2
	c_2	e_1		
a_2	c_2	e_1, e_3	g_1	h_1
			g_1	h_2
a_3	c_3	e_1	g_2	h_3

Figure 3. The relation x_1.

A selection condition is a boolean expression involving comparisons between attributes and between attributes and constants. Comparison operators also include set comparison and set membership operators, $\subset, \subseteq, =, \supseteq, \supset, \in$, and $\notin$ in addition to the usual comparison operators, $<, \leq, =, \neq, \geq$, and $>$. Let R be a relation scheme. Then, L is a 'select list' of R if (i) L is empty (ii) L is of the form $(R_{1c_1}L_1, R_{2c_2}L_2, ..., R_{nc_n}L_n)$, $(1 \leq n \leq |\ RAttr(R)\ |)$ where each R_i is a relation-valued attribute of R, c_i is a condition on R_i, L_i is a select list of R_i, and $R_i \neq R_j$ if $i \neq j$ $\forall R_{jc_j}L_j \in L$.

Let r be a relation with relation scheme R. Let L be a select list of R and let c be a condition on R. Then $\sigma(r_cL)$ is defined as follows.

$$
\begin{aligned}
&(i)\quad \sigma(r_c) = \{t \in r \mid c(t) = true\}\\
&(ii)\quad \sigma(r_c(R_{1c_1}L_1, R_{2c_2}L_2, ..., R_{nc_n}L_n))\\
&\qquad = \{t \mid (\exists t_r \in r) \wedge\\
&\qquad\quad (t[Attr(R) - \{R_1, R_2, ..., R_n\}]\\
&\qquad\qquad = t_r[Attr(R) - \{R_1, R_2, ..., R_n\}])\\
&\qquad\quad \wedge (c(t_r) = true)\\
&\qquad\quad \wedge (t[R_1] = \sigma\left((t_r[R_1])_{c_1}L_1\right) \neq \phi)\\
&\qquad\quad \vdots\\
&\qquad\quad \wedge (t[R_n] = \sigma\left((t_r[R_n])_{c_n}L_n\right) \neq \phi)\}
\end{aligned}
$$

Figure 4(a) shows an example of a recursive selection, $\sigma(x_1(B_{C=c_2}))$, on the relation x_1, shown in Figure 3.

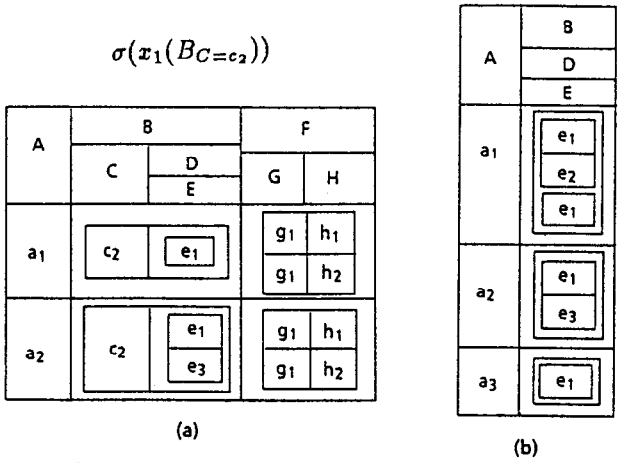

Figure 4. Examples of recursive selection and projection on x_1.

Projection π

The recursive definition of projection given here allows projections to be performed on attributes at all levels without restructuring.

Let R be a relation scheme. Then, L is a 'project list' of R if (i) L is empty (ii) L is of the form $(R_1L_1, ..., R_nL_n)$ where each R_i is an attribute of R, L_i is a project list of R_i (L_i is empty if R_i is an atomic-attribute), and $R_i \neq R_j$ if $i \neq j$ $\forall R_jL_j \in L$.

Let r be a relation with relation scheme R and let L be a project list of R. Then $\pi(Lr)$ is defined as follows.

(i) $\pi(r) = r$

(ii) $\pi((R_1L_1, ..., R_nL_n)r)$

$$= \{t \mid (\exists t_r \in r) \wedge (t[R_1] = f(t_r, R_1L_1)) \wedge$$
$$\vdots$$
$$\wedge (t[R_n] = f(t_r, R_nL_n))\}$$

where $f(t_r, R_iL_i) = t_r[R_i]$ if $R_i \in FAttr(R)$

$= \pi(L_i(t_r[R_i]))$ if $R_i \in RAttr(R)$

Figure 4(b) shows an example of a recursive projection, $\pi(A, B(D))$, on x_1.

Join $\bowtie$

Joins, like all the other operators are defined recursively so that a relation can be joined to another relation or any subrelation of the relation.

Let R be a relation scheme. L is a join-path of R if (i) L is empty. (ii) L is of the form (R_iL_i) where R_i is a relation-valued attribute of R and L_i is a join-path of R_i.

Let r and q be two relations with relation schemes R and Q respectively and let L be a join-path of R. Then $\bowtie(rL, q)$ is defined as follows.

(i) $\bowtie(r, q)$ =

$\{t \mid (\exists t_r \in r, t_q \in q)$

$\wedge (t[R_1, ..., R_n] = t_r[R_1, ..., R_n] = t_q[R_1, ..., R_n])$

$\wedge (t[Attr(R) - \{R_1, ..., R_n\}] =$

$t_r[Attr(R) - \{R_1, ..., R_n\}])$

$\wedge (t[Attr(Q) - \{R_1, ..., R_n\}] =$

$t_q[Attr(Q) - \{R_1, ..., R_n\}])\}$

where $\{R_1, ..., R_n\}$ is the set of common attributes of R and Q

(ii) $\bowtie(r(R_iL_i), q)$ =

$\{t \mid (\exists t_r \in r)$

$\wedge (t[Attr(R) - \{R_i\}] = t_r[Attr(R) - \{R_i\}])$

$\wedge (t[R_i] = \bowtie(t_r[R_i]L_i, q) \neq \phi)\}$

Figure 5 shows an example of a join between a subrelation of a relation and a relation.

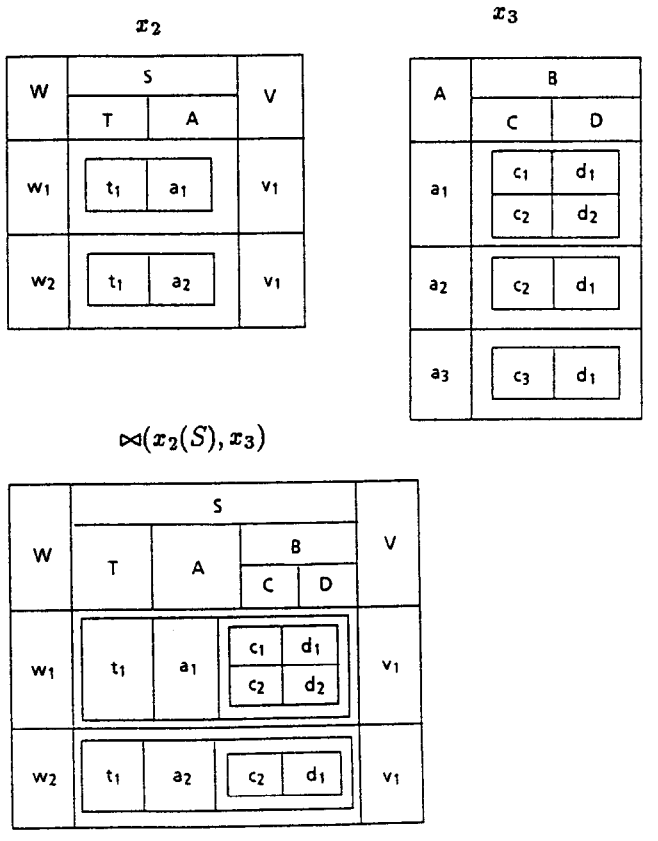

Figure 5. An example of a recursive join.

The definitions for cartesian product, nest, unnest, union, difference, and intersection, which are also recursive, are described in [5], but omitted here for brevity. Even though the recursive definitions for the operators allow each operator to apply itself recursively to tuples at all levels without restructuring, the restructuring operators nest and unnest are, however, necessary since users may wish to view the data in a different format.

2.2 A Comparison of the Recursive and Non-Recursive Algebras

As mentioned earlier all the operators of the non-recursive algebra, *i.e.*, the algebra of Thomas and Fischer, except nest and unnest, have definitions that are very similar to the definitions in the relational algebra. Nest and unnest are restructuring operators which add another level of nesting to a relation and flatten a relation by one level respectively. Since all the non-recursive operators can access and modify only tuples at the outermost level of a relation, the unnest operator is used to flatten the relation so that these operators can be applied to the flattened relation. The nest operator is then used to bring the relation back into its original form. We give the definitions for nest and unnest, as defined in the non-recursive algebra, before considering some example queries to see how they are expressed in the recursive and the non-recursive algebras.

Nest ν

The nest operator, also sometimes called 'pack' [14], groups together tuples which agree on all the attributes that are not in a given set of attributes, say B. It forms a single tuple, for every such group, which has a new attribute, say A, in place of B, whose value is the set of all the B values of the tuples being grouped together.

$$\nu_{B\rightarrow A}(r) = \{t \mid (\exists u \in r) \wedge (t[Attr(R) - B] = u[Attr(R) - B]) \wedge (t[A] = \{s[B] \mid (s \in r) \wedge (s[Attr(R) - B] = u[Attr(R) - B])\})\}$$

where $B \subseteq Attr(R)$ and A is a new attribute name

Unnest μ

The unnest or 'unpack' operator does the inverse of the nest operator by 'ungrouping' or flattening out the B value of the tuples.

$$\mu_B(r) = \{t \mid (\exists u \in r) \wedge (u[Attr(R) - \{B\}] = t[Attr(R) - \{B\}]) \wedge (t[Attr(B)] \in u[B])\} \quad \text{where} B \in RAttr(R)$$

Example 1

Consider the example database shown in Figure 2. Let us suppose that we want the set of all the shares purchased on 02/10/83 and we want them listed by their owner's name and the company of the share. We would formulate this query in the recursive algebra as follows:

π((NAME,INVESTMENTS)
(σ(CLIENTS(INVESTMENTS ($SHARES_{DATE='02/10/83'}$)))))

Figure 6 shows the result of this query. In the non-recursive algebra, the same query would have to be expressed using nest and unnest as shown below.

$\nu_{COMPANY,SHARES\rightarrow INVESTMENTS}$
($\nu_{PURCHASE-PRICE,DATE,NO.\rightarrow SHARES}$
($\pi_{NAME,COMPANY,PURCHASE-PRICE,DATE,NO.}$
($\sigma_{DATE='02/10/83'}$(μ_{SHARES}
($\mu_{INVESTMENTS}$(CLIENTS))))))

The two unnests transform the relation CLIENTS into a flat relation. Then, after the selection and projection have been performed on the flat relation, the nest operations, restructure the flat relation back into a nested one.

NAME	INVESTMENTS			
	COMPANY	SHARES		
		PURCHASE PRICE	DATE	NO.
John Smith	XEROX	64.50	02/10/83	100
Jill Brody	EXXON	59.50	02/10/83	200
	FORD	35.50	02/10/83	200

Figure 6. The result of the query in Example 1.

Example 2

Let us suppose that we want the investments that clients have in stock that is traded in London, listed by the client's name and address. This query, whose result is shown in Figure 7, can be expressed in the non-recursive algebra, using the recursive join which allows joins to be performed between a subrelation and a relation as follows.

$\bowtie$(CLIENTS (INVESTMENTS) ,π((COMPANY)
(σ(STOCK-DATA$_{LONDON \in EXCHANGES-TRADED}$))))

NAME	ADDRESS	INVESTMENTS			
		COMPANY	SHARES		
			PURCHASE PRICE	DATE	NO.
John Smith	311 East 2nd. St. Bloomington, IN 47401	IBM	89.75	06/20/83	200
			96.50	11/10/84	100
Jill Brody	41 North Main St. Oberlin, OH 44074	EXXON	35.00	01/30/81	100
			64.50	01/30/82	100
			59.50	02/10/83	200

Figure 7. The result of the query in Example 2.

The same query would be expressed in the non-recursive algebra as follows.

$$\nu_{\text{COMPANY,SHARES}\rightarrow\text{INVESTMENTS}}(\mu_{\text{INVESTMENTS}}(\text{CLIENTS}) \bowtie \pi_{\text{COMPANY}}(\sigma_{\text{LONDON}\in\text{EXCHANGES-TRADED}}(\text{STOCK-DATA})))$$

Although the expressions in the non-recursive algebra, for the above examples, work in this particular case, in general relations have to be *indexed* first before unnesting and nesting. These expressions may not give us the correct result if the relations in Figure 2 were not in Partitioned Normal Form (a relation is in Partitioned Normal Form (PNF) if (i) all or a subset of the atomic attributes form a key for the relation and (ii) every subrelation of the relation is in PNF). In [26] Van Gucht has shown that if r is a relation with relation scheme R and A is a relation-valued attribute of R, then

$$\nu_{Attr(A)\rightarrow A}(\mu_A(r)) = r$$

if and only if r satisfies the functional dependency

$$Attr(R) - \{A\} \rightarrow A$$

In [28], Van Gucht and Fischer define an *Index* operator, which is basically a way of tagging the tuples of a relation to preserve the information about how tuples were grouped together before unnesting. A slightly modified definition of the *Index* operator is given below.

Definition

$Index(r) = \{t \mid (\exists t_r \in r) \wedge (t[Attr(R)] = t_r[Attr(R)]) \wedge (t[I] = \{t_r\})\}$

This operator can be expressed in terms of nest, selection and cartesian product. $Index(r) = \nu_{A'\rightarrow I}\sigma_{A'=A}(r \times r)$, where A is the set of attributes of R and A' is the set of new attributes in $r \times r$. Tagging the tuples of a relation is required to save information about how tuples were grouped together before an unnest was performed.

The correct expression for the query in Example 1 is then

$$\pi_{\text{NAME,INVESTMENTS}}(\nu_{\text{COMPANY,SHARES}\rightarrow\text{INVESTMENTS}}(\pi_{\text{NAME,COMPANY,SHARES,}I_2}(\nu_{\text{PURCHASE-PRICE,DATE,NO.}\rightarrow\text{SHARES}}(\pi_{\text{NAME,COMPANY,PURCHASE-PRICE,DATE,NO.,}I_1,I_2}(\sigma_{\text{DATE='02/10/83'}}(\mu_{\text{SHARES}}(Index(\mu_{\text{INVESTMENTS}}(Index(\text{CLIENTS}))))))))))$$

where, I_1 and I_2 are the index columns added by the two *Index* operations.

In some cases, even indexing a relation before unnesting will not give back the original relation after nesting. Unnesting on a relation-valued attribute that contains empty sets as values causes those tuples to be lost. In such cases expressions for queries that involve unnesting become even more complicated since special consideration must be given to such tuples. Null values can be used to overcome this problem and the interested reader is refered to [21] for a discussion of null values in the context of the NF^2 model.

Thus, queries in the non-recursive algebra are long and complicated and cause data to be restructured while performing operations like selection and projection. This restructuring is necessary because the algebraic operators operate only at the outermost level of a relation. Hence, although the first normal form assumption has been relaxed to include relation-valued attributes, the algebraic operators essentially treat all the components of a tuple as non-decomposable or atomic units. The recursive algebra, on the other hand, has operators that can apply themselves recursively to the subrelations of nested relations. Queries are processed much more efficiently when expressed in the recursive algebra since restructuring is not necessary for accessing the subrelations of a relation. They are also much more succinct than the corresponding queries in the non-recursive algebra and allow users to remain within the framework of the model for complex objects while querying the database, by obviating the need for restructuring these objects during the querying process.

Theorem

The recursive and the non-recursive algebras are each expressible in terms of each other and are hence equivalent to each other in terms of their expressive powers.

From the definitions of the recursive operators, it is obvious that the non-recursive operators can be easily expressed in terms of the recursive operators. Each recursive operator can be expressed in the non-recursive algebra by applying a sequence of unnests until the attributes of interest are at the outermost level, applying

the operator, and then applying a sequence of nests. As mentioned earlier, if a relation is not in PNF, it must be indexed before every unnest. Also, unnesting causes tuples that contain empty sets as values (for the relation-valued attribute on which the unnest is performed) to be lost. These tuples must be accounted for while expressing a recursive operator in the non-recursive algebra (except in the case of selection). In the case of difference tuples can be lost not only as a result of performing an unnest on an attribute containing empty sets as values, but also when taking the difference after unnesting. The proof for the equivalence theorem can be found in [5].

3. Query Optimization

Optimizing an expression denoting a query involves determining an equivalent expression for that query which can be processed in lesser time than the original expression. Finding common subexpressions, performing selections and projections before joins, combining several selections into one selection, etc., are some of the strategies for finding equivalent but more efficient expressions for queries in the relational algebra. Laws about the commutativity and the associativity of operators are used to rearrange the operators in order to find equivalent expressions. Most of these laws which apply to the operators of the relational algebra also apply to the recursive operators for nested relations. The only exceptions are the commutativity laws for joins and selections when these operations involve subrelations. We list some of these properties for the recursive algebra.

We first define *merge*, *common−merge*, and *join−lists*

Definitions:

(*i*) Let r be a relation with relation scheme R and let $L_1 = (R_{a_1 c_{a_1}} L_{a_1}, ..., R_{a_n c_{a_n}} L_{a_n})$ and $L_2 = (R_{b_1 c_{b_1}} L_{b_1}, ..., R_{b_m c_{b_m}} L_{b_m})$ be select lists of R.

$merge(L_1, L_2) = L_1$ if L_2 is empty
$= L_2$ if L_1 is empty
$= L$ otherwise, where L is such that

$R_{a_i c_{a_i}} L_{a_i} \in L$ if $\forall R_{b_j c_{b_j}} L_{b_j} \in L_2, R_{b_j} \neq R_{a_i}$
$R_{b_i c_{b_i}} L_{b_i} \in L$ if $\forall R_{a_j c_{a_j}} L_{a_j} \in L_1, R_{a_j} \neq R_{b_i}$
$R_{a_i (c_{a_i} \wedge c_{b_j})} merge(L_{a_i}, L_{b_j}) \in L$ if $R_{b_j} = R_{a_i}$ for some $R_{a_i c_{a_i}} L_{a_i} \in L_1$, $R_{b_j c_{b_j}} L_{b_j} \in L_2$
nothing else is in L

If L_1 and L_2 were project lists instead of select-lists, the result of $merge(L_1, L_2)$ would be very similar.

(*ii*) If $L_1 = (R_{a_1} L_{a_1}, ..., R_{a_n} L_{a_n})$ and $L_2 = (R_{b_1} L_{b_1}, ..., R_{b_n} L_{b_m})$ are project lists of R, then

$common - merge(L_1, L_2)$
$= L_1$ if L_2 is empty
$= L_2$ if L_1 is empty
$= L$ otherwise, where L is such that
R_{a_i} $common - merge(L_{a_i}, L_{b_j}) \in L$ if $R_{b_j} = R_{a_i}$ for some $R_{a_i} L_{a_i} \in L_1$, $R_{b_j} L_{b_j} \in L_2$
nothing else is in L

(*iii*) If L_1 is a join path of R and L_2 is a project-list or join-path of R, then

$join - lists(L_1, L_2) = L_1$ if L_2 is empty
$= L_2$ if L_1 is empty
$= (R_i\ join - lists(L_i, L_2))$ if L_1 is of the form $(R_i L_i)$

Algebraic Equivalences:

Let c_1 and c_2 be two conditions on R and let L_1 and L_2 be two select lists on R. Then,

1. $\sigma((\sigma(r_{c_1}))_{c_2}) = \sigma(r_{(c_1 \wedge c_2)})$
2. $\sigma((\sigma(r_{c_1} L_1))_{c_2} L_2) =$
$\sigma((\sigma(r_{(c_1 \wedge c_2')} merge(L_1, L_2')))_{c_2''} L_2'')$
where $c_2 = c_2' \wedge c_2''$, c_2' contains only attributes that are not in L_1 and L_2 can be split up into two lists L_2' and L_2'' such that L_2' contains only attributes that are not in L_1 and not in c_2''.
3. If L_1 and L_2 are project lists on R, then
$\pi(L_1(\pi L_2(r))) = \pi(common - merge(L_1, L_2)r)$
4. If L_1 is a project list on R, L_2 a select list on $\pi(L_1 r)$ and c a condition on R, then
$\sigma((\pi(L_1 r))_c L_2) = \pi(L_1(\sigma(r_c L_2)))$
provided that $\pi(L_1 r)$ does not change the structure of the attributes in c and L_2.
5. If L_1 is a project list on R, L_2 a select list on R

and c a condition on R, then

$$\pi(L_1(\sigma(r_c L_2))) = \sigma((\pi(L_1 r))_c L_2)$$

if c and L_2 do not contain any attributes that are not in L_1 and provided that $\pi(L_1 r)$ does not change the structure of the attributes in c and L_2.

6. If L_2 and c are the same as above but L_1 is a join path of R, then

$$\sigma((\bowtie(rL_1, q))_c L_2) = \bowtie(\sigma(r_c L_2)L_1, q)$$

if c and L_2 contain no attributes in common with L_1 and q.

7. Let r, q, and p be relations with schemes R, Q, and P respectively.

(a) $\bowtie(r, q) = \bowtie(q, r)$

(b) $\bowtie((\bowtie(r, q)), p) = \bowtie(r, \bowtie(q, p))$

(c) $\bowtie(rL, q) \neq \bowtie(qL, r)$ where L is a non-empty join-path of r.

(d) If L_1 is a join path of R, then

$\bowtie(\bowtie(rL_1, q)L_2, p) =$

$\bowtie(rL_1, \bowtie(qL_2', p))$ if L_2 is a join-path to an attribute of q in $\bowtie(rL_1, q)$ where L_2' is the join-path in q to that attribute and that attribute is not involved in the join between r and q.

$\bowtie(\bowtie(rL_2, p)L_1, q)$ if L_2 is a join-path to an attribute in r and that attribute is not involved in the join between r and q.

8. If L_2 is a join path of R and L_1 is a project list of $\bowtie(rL_2, q)$, then

$$\pi(L_1(\bowtie(rL_2, q))) = \pi(L_1(\bowtie(\pi(L_1' r)L_2, \pi(L_2' q))))$$

if L_1 can be split up into L_1' and L_2' such that they contain attributes of r and q in L_1 respectively, and they each also contain all common attributes involved in the join.

9. If L_1 is a project list of R and L_2 is a join path of $\pi(L_1 r_1)$, then

$\bowtie(\pi(L_1 r)L_2, q) =$
$\pi(merge(L_1, join\text{-}lists(L_2, Attr(q)))(\bowtie(rL_2, q)))$

if L_1 contains all the common attributes between r and q and if $\pi(L_1 r)$ does not change the structure of these common attributes.

As shown above, most of the commutative and associative laws for the recursive operators are a natural extension of the laws for the operators of the relational algebra. Even though these laws also apply to the operators of the non-recursive algebra for nested relations, since the operators nest and unnest do not commute with most operators, queries that contain these operators are not easily optimized. Consider the two expressions in the non-recursive algebra given below, where r_1, r_2, and r_3 are the relations shown in Figure 8.

(*i*) $\nu_{A \to A'}((\mu_{A'}(\nu_{D \to C}(\pi_{A',B,D,F}(\mu_C(r_1 \bowtie r_2))))) \bowtie r_3)$
(*ii*) $\nu_{D \to C}(\pi_{A',B,D,F}(\mu_C((\nu_{A \to A'}(\mu_{A'}(r_1) \bowtie r_3)) \bowtie r_2)))$

r_1

A'	B	C	
A		D	E
a1	b1	d1	e1
a2		d2	e1
a2	b2	d1	e1
		d2	e1
a2	b3	d2	e2
a3			
a2	b4	d3	e1

r_2

C		F
D	E	
d1	e1	f1
d2	e1	
d2	e2	f2
d2	e2	f3
d3	e1	f4

r_3

A
a1
a4

r_4

A'	B	C	F
A		D	
a1	b1	d1	f1
		d2	

Figure 8.

The relation r_4 in Figure 8 shows the result of these equivalent expressions. If r_3 is likely to have far lesser matches with r_1 as compared to r_2, then the second expression will be more efficient than the first. But since join does not commute with nest, it is not possible to derive the second expression from the first by applying commutative and associative laws on the operators.

On the other hand, in the recursive algebra, this problem does not often arise since nest and unnest are not used very often in queries, as they are not needed for accessing subrelations but are used only when the data is restructured. Now, let us consider the equivalent expressions in the recursive algebra for the ones in the above example.

(i) $\bowtie(\pi((A', B, C(D), F)(\bowtie(r_1, r_2)))(A'), r_3)$
(ii) $\pi((A', B, C(D), F)(\bowtie(\bowtie(r_1(A'), r_3), r_2)))$

The second expression can be easily derived from the first by applying laws, 9 and 7d.

So, since queries in the non-recursive algebra are typically long and complicated with a number of nests and unnests, they are not easily optimized. The same queries can be optimized more efficiently in the recursive algebra, since nest and unnest are used only when restructuring is really necessary.

4. Conclusions

The nested relational model is more suitable for representing complex objects than the traditional relational model. However, most query languages that have been proposed for this model require either restructuring operators or special navigational operators for accessing tuples that are nested at different levels in a nested relation. In this paper, we have described a recursive algebra for nested relations which allows queries to be expressed succinctly without any restructuring or special navigational operators. The nested relational model, is a recursive extension of the relational model; it is hence only natural that query languages for this model be extended similarly. We have also developed query optimization principles for this algebra and shown that most of them can be viewed as extensions of the optimization principles for the relational algebra. Finally, we have pointed out the problems in optimizing queries written in query languages which require restructuring operations in order to access subrelations. In conclusion, we claim that the recursive algebra is better suited for the nested relational model than other query languages that have been proposed for this model.

Acknowledgements

I would like to thank Marc Gyssens, Dirk Van Gucht and the refrees for their helpful suggestions and comments.

References

1. Abiteboul, S., and Bidoit, N., "Non First Normal Form Relations to Represent Hierarchically Organized Data," *Proc. 3rd PODS*, 1984, pp. 191-200.
2. Bancilhon, F., Richard, P., and Scholl, M., "On Line Processing of Compacted Relations," *Proc. 8th VLDB*, Mexico City, 1982, pp. 21-37.
3. Bidoit, N., "The Verso Algebra or How to Answer Queries with Fewer Joins," *Journal of Computer and System Sciences*, 1987, pp. 321-364
4. Codd, E.F., "A Relational Model for Large Shared Data Banks, " *Communications ACM* Vol. 6, No. 13, June 1970, pp. 377-387.
5. Colby, L.S., "A Recursive Algebra for Nested Relations," *Technical Report, No. 259*, August 1988, University of Indiana
6. Dadam, P., Kuespert, F., Andersen, F., Blanken, H., Erbe, R., Guenauer, J., Lum, V., Pistor, P., and Walch,G., "A DBMS Prototype to Support Extended NF2 Relations: An Integrated View on Flat Tables and Hierarchies," *Proc. Annual SIGMOD Conf.*, Austin, 1986, pp. 356-366.
7. Deppisch, U., Paul, H.B., and Schek, H.J., "A Storage System for Complex Objects," *Proc. Int. Workshop on Object- Oriented Database Systems*, Pacific Grove, 1986, pp. 183-195.
8. Deshpande, V., and Larson, P.A., "An Algebra for Nested Relations," *Tech. Report, CS-87-65*, 1987, University of Waterloo.
9. Deshpande, A., and Van Gucht, D., "A Storage Structure for Unnormalized Relations," *Proc. GI Conf. on Database Systems for Office Automation, Engineering and Scientific Applications*, Darmstadt, April 1987, pp. 481-486.
10. Gyssens, M. and Van Gucht, D., "The Powerset Algebra as a Result of Adding Programming Constructs to the Nested Relational Algebra," *Proc. Annual SIGMOD Conf.*, Chicago, IL, June 1988
11. Jaeshke, G., and Schek, H.J., "Remarks on the Algebra on Non-First Normal Form Relations," *Proc. 1st PODS*, Los Angeles, 1982, pp. 124-138.
12. Linnemann, V., "Non First Normal Form Relations and Recursive Queries: An SQL-Based Approach," *Proc. 3rd IEEE Int. Conf. on Data Engineering*, Los Angeles, 1987.
13. Makinouchi, A., "A Consideration of Normal Form of Not- Necessarily-Normalized Relations in the Relational Data Model," *Proc. 3rd. VLDB*, Tokyo 1977, pp. 447-453.
14. Ozsoyoglu, G., Ozsoyoglu, Z.M., and Matos, V., "Extending Relational Algebra and Relational Calculus with Set-Valued Attributes and Aggregate Functions," *ACM Transactions on Database Systems*, Vol. 12, No. 4, December 1987, pp. 566-592.
15. Paul, H.B., Schek, H.J., Scholl, M.H., Weikum, G., and Deppisch, U., "Architecture and Implementation of the Darmstadt Database Kernel Sys-

tem," *Proc., Annual SIGMOD Conf.*, San Francisco , 1987, pp. 196-207.

16. Pistor, P., and Andersen, F., "Designing a generalized NF2 model with an SQL-Type Language Interface," *Proc. 12th VLDB,* Kyoto, Japan, 1986, pp. 278-288.
17. Pistor, P., and Traunmueller, R., "A Database Language for Sets, Lists and Tables," *Information Systems*, Vol 11, No. 4, 1986, pp. 323-336.
18. Roth, M.A., Korth, H.F., and Batory, D.S., "SQL/-NF: A Query Language for ¬1NF Relational Databases," *Information Systems*, Vol 12, No. 1, 1987, pp. 99-114.
19. Roth, M.A., Korth, H.F., and Silberschatz, A., "Theory of Non-First-Normal-Form Relational Databases," *Tech. Report TR-84-36 (Revised January 1986)*, University of Texas at Austin, 1984.
20. Roth, M.A., Korth, H.F., and Silberschatz, A., "Extended Algebra and Calculus for ¬1NF Relational Databases," *Tech. Report TR-85-19*, University of Texas at Austin, 1985.
21. Roth, M.A., Korth, H.F., and Silberschatz, A., "Null Values in ¬1NF Relational Databases," *Tech. Report TR-85-32*, University of Texas at Austin, December, 1985.
22. Schek, H.J., and Scholl, M.H., "The Relational Model with Relation-Valued Attributes," *Information Systems*, Vol. 11, No. 2, 1986, pp. 137-147.
23. Scholl, M.H., "Theoretical Foundation of Algebraic Optimization Utilizing Unnormalized Relations," *Proceedings of the International Conference on Database Theory*, September, 1986, Rome, Italy, pp. 380-396
24. Scholl, M.H., Paul, H.B., and Schek, H.J., "Supporting Flat Relations by a Nested Relational Kernel," *Proc. 13th VLDB*, London, 1987.
25. Thomas, S.J., and Fischer, P.C., "Nested Relational Structures," *Advances in Computing Research III, The Theory of Databases*, P.C. Kanellakis, ed., JAIpress, 1986, pp. 269-307.
26. Van Gucht, D., "Theory of Unnormalized Relational Structures," *Ph.D. Dissertation*, Vanderbilt University, 1985.
27. Van Gucht, D., "On the Expressive Power of the Extended Relational Algebra for the Unnormalized Relational Model," *Proc. 6th PODS*, San Diego, CA, March 1987, pp. 302-312.
28. Van Gucht, D., and Fischer, P.C., "Multilevel Nested Relational Structures," *Journal of Computer and System Sciences*, Vol. 36, No. 1, February 1988, pp. 77-105