TECHNICAL REPORT NO. 322

Optimization of Nonmonotonic Relational Queries

by

Lawrence V. Saxton and Dirk Van Gucht

December 1990

COMPUTER SCIENCE DEPARTMENT
INDIANA UNIVERSITY
Bloomington, Indiana 47405-4101

Optimization of Nonmonotonic Relational Queries

Lawrence V. Saxton* and Dirk Van Gucht
Computer Science Department
Indiana University
Bloomington Indiana 47405
email: vgucht@iuvax.cs.indiana.edu

Abstract

In the relational database model, the nonmonotonic queries include the relational algebra operators of difference, quotient, aggregation and complement, as well as certain relational calculus queries involving universal quantification and negation. It is known that for nested algebra queries whose input and output are standard relational databases, there is an equivalent relational algebra query. We show that the nonmonotonic queries can be handled naturally as such set queries expressed in the nested algebra. We then show that these queries are efficiently implementable even in a standard relational database. This relies on efficient $O(n\,log\,n)$ ordering algorithms for operations involving set constructions and comparisons.

1 Introduction

The popularity of the commercial relational database systems is in great part due to the efficiency with which the basic monotonic operators, select, project and join, have been implemented. However, as noted by Date, [7], the maturation of the use of these systems commercially has led to the usage of decision support queries. These queries are more apt to require nonmonotonic queries in their formulation, [2, 7].

The nonmonotonic queries have the property that if the database is increased in size the application of the query may actually result in a smaller output. Consider a banking application where there is to be a promotion for a new investment account. The bank wants to mail a flyer to all customers who have both savings and checking accounts, but who

*On leave from the University of Regina

have no personal loans. This decision support query is nonmonotonic since adding a tuple to reflect the fact that a customer took a personal loan would cause that customer to be removed from the output for the query.

Many researchers, [4, 6, 8, 9, 10, 12, 18, 19, 20, 21, 26, 27, 28, 30], have shown that these decision support queries are more readily expressible by using forms of intermediate set construction and comparison. However, some of these studies limit their discussion to specific nonmonotonic queries, [6, 8, 10, 27, 28]. Others, including [4, 9, 12, 13, 14, 18, 30], use suboptimal iterative techniques to compute their results or translate them to the underlying relational operators. The final group, [3, 19, 20, 21], deal with an extended algebra which allows for complex objects (generally simple sets or aggregations) to be used in the input database. Unfortunately, the number of sets that can be so represented could be large (exponentially so in terms of the number of elementary values), which in turn increases the complexity of their results.

Although in decision support queries both the input and output conform to relational databases, in general, they require the use of some temporary internal sets as part of their computation, [19, 20]. Paredaens and Van Gucht, [22], recently have shown that queries of this type can be expressed equivalently in either the standard relational formats or as nested relational queries, called *flat-flat queries*. Hence, we know that at any level of set construction internally, the computation time for a query as a whole is polynomially bounded in terms of the number of elementary values of the input database.

The goal of this research is to provide an uniform approach for handling nonmonotonic queries as nested flat-flat queries and to show the efficiency with which this can be achieved with standard relational databases.

The paper is organized as follows. In Section 2, we provide the necessary basic definitions. In Section 3, we introduce the nonmonotonic queries and show their relationship to queries involving set operations. Next, these set operations and their related orderings are shown to be efficiently implementable in Section 4. The conclusions and extensions are discussed in the final section.

2 Basic Definitions

Since nested relations are more general than standard relations, we will state our definitions in terms of nested relations. Our definitions are based on [2].

2.1 Nested Relations

Assume an infinitely enumerable set U of *elementary attributes*. Attributes are either elementary or composite, where a composite attribute is a set of elementary or composite attributes.

Definition 1 *The set of all attributes $\mathcal{U}$ is the smallest set containing U such that every finite subset X of $\mathcal{U}$, in which no elementary attribute appears more than once, is in $\mathcal{U}$.*

Elements of U are called *elementary*; those of $\mathcal{U} - U$ are called *composite* (or relation-valued). We denote elementary attributes by $A, B, C, \ldots$, composite attributes by $X, Y, Z, \ldots$.

Next we define simultaneously the notions of value, tuple and instance. Assume an infinitely enumerable set V of *elementary values*.

Definition 2 *The set $\mathcal{V}$ of all values, the set $\mathcal{I}_X$ of all instances over $X \in \mathcal{U} - U$, the set $\mathcal{T}_X$ of all tuples over $X \in \mathcal{U} - U$, and the set $\mathcal{I}$ of all instances are the smallest sets satisfying:*

- $\mathcal{V} = V \cup \mathcal{I}$;
- $\mathcal{I} = \bigcup_{X \in \mathcal{U} - U} \mathcal{I}_X$;
- *$\mathcal{I}_X$ consists of all finite subsets of $\mathcal{T}_X$;*
- *$\mathcal{T}_X$ consists of all mappings t from X into $\mathcal{V}$, called tuples, such that $t(A) \in V$ for all $A \in X \cap U$ and $t(Y) \in \mathcal{I}_Y$ for all $Y \in X - U$.*

Definition 3 *A (nested) relation is a pair (Ω, ω) where $\Omega \in \mathcal{U} - U$ and $\omega \in \mathcal{I}_\Omega$. Ω is called the scheme of the relation and ω is called the instance of the relation. The set of all relations with scheme Ω will be denoted by $\mathcal{R}_\Omega$, and the set of all relations will be denoted by $\mathcal{R}$.*

Assume an infinitely enumerable set N of *relation names*. A (nested) *database scheme* is a sequence $\Delta = [R_1 : \Omega_1, \ldots, R_n : \Omega_n]$, where $R_i \in N$, and Ω_i is a relation scheme. A (nested) *database instance* over Δ is a sequence $\delta = [R_1 : \omega_1, \ldots, R_n : \omega_n]$, where $\omega_i \in \mathcal{I}_{\Omega_i}$. The set of databases instances over Δ is denoted $\mathcal{I}_\Delta$. A (nested) *database* is a pair (Δ, δ), where $\delta \in \mathcal{D}_\Delta$. The set of all databases over Δ will be denoted by $\mathcal{D}_\Delta$. The set of all databases will be denoted by $\mathcal{D}$.

The traditional (non-nested) relational model consists of the restriction of relation schemes to sets of elementary attributes. In the sequel, we will call the traditional relational model the *standard* relational model, and refer to the above defined concepts in its context with the adjective standard.

As a matter of notational simplicity throughout the paper, we will denote composite attributes as made up of their elementary attribute names using a superscript $*$ to define sets. For example, a composite attribute denoted $(A^*B)^*$ could contain tuples of the form $[\{a_1, a_2\}b_1]$.

Example 1 Consider the standard relational database, R_a, shown in Figure 1(a). Each of the attributes, (A, B and C), are elementary. In Figure 1(b), a nested relational database is shown. There are three attributes, B and C are elementary attributes, while A^* is a composite attribute which has simple set entries. In Figure 1(c), we show a more complex nested relational database, with elementary attribute C and two composite attributes A^* and B^*. $\Box$

2.2 Queries

The *active domain* of a tuple t (relation instance ω, database instance δ), denoted $adom(t)$ ($adom(\omega)$, $adom(\delta)$), is the set of elementary values appearing in t (ω, δ).

A	B	C
1	1	1
2	1	1
1	2	1
2	2	1
3	3	2
4	3	2

$(a) R_a(ABC)$

A^*	B	C
$\{1,2\}$	1	1
$\{1,2\}$	2	1
$\{3,4\}$	3	2

$(b) R_b(A^*BC)$

A^*	B^*	C
$\{1,2\}$	$\{1,2\}$	1
$\{3,4\}$	$\{3\}$	2

$(c) R_c(A^*B^*C)$

Figure 1: Example of (Nested) Relations

Definition 4 *A query from a database scheme Δ_{in} to a database scheme Δ_{out}, denoted $q : \Delta_{in} \rightarrow \Delta_{out}$*[1]*, is a (partial) mapping from $\mathcal{D}_{\Delta_{in}}$ to $\mathcal{D}_{\Delta_{out}}$, such that for some finite set $C \subset V$ the following holds:*

- *q is domain preserving w.r.t C: $\forall \delta \in \mathcal{I}_{\Delta_{in}}$, $adom(q(\delta)) \subseteq adom(\delta) \cup C$;*
- *q is C-generic: for each permutation σ over V (extended in the natural way to $\mathcal{I}$ and $\mathcal{V}$), such that $\forall c \in C$, $\sigma(c) = c$, $q \circ \sigma = \sigma \circ q$.*

In addition, we require the following definition of monotonicity adapted from [2].

Definition 5 *A query $q : \Delta_{in} \rightarrow \Delta_{out}$ is said to be monotone if for all $S, R \in \mathcal{I}_{\Delta_{in}}$, $S \subseteq R$, then $q(S) \subseteq q(R)$. Otherwise, a query is said to be nonmonotone.*

2.3 Relational query languages

We will consider several relational query languages. In particular, for relational databases we will consider the standard relational algebra, the standard relational calculus [5] and nested SQL, [8]. For nested relational databases we will consider the nested algebra and the nested calculus [24].

[1]In the case where Δ_{out} consists of a single component $[R_{out} : \Omega_{out}]$, we will also talk about a query from a database scheme Δ_{in} to a relation scheme Ω_{out}.

2.4 Algebraic query languages

Algebraic query languages for (nested) relations are obtained by extending the standard relation algebra operators to deal with (nested) relations, and adding the restructuring operators unnest and nest. Hence, the operators are:

Definition 6

- *The classical relational operators of union ($\cup$), difference ($-$), division ($\div$), cartesian product ($\times$), join ($\bowtie$), and projection (π). Cartesian product is applicable only to relations whose schemes are built from disjoint sets of elementary attributes. Required renaming of attributes is performed by the rename operator. If (Ω, ω) is a relation and T an attribute in Ω then the renaming of T by T' in (Ω, ω) is denoted $\rho_{T \to T'}(\Omega, \omega)$.*

- *Selection of tuples from a relation (Ω, ω) is defined relative to a predicate Ψ on tuples as $\sigma_\Psi(\Omega, \omega) = (\Omega, \{t : t \in \omega \wedge \Psi(t) = \mathtt{true}\})$. We consider the following predicates: Elementary attribute comparison, $A\Theta B$, for A, $B \in \Omega \cap U$; Elementary attribute-constant comparison, $A\Theta a$, for $A \in \Omega \cap U$ and $a \in V$, and Θ is either $=, \neq, <, \cdots$; Composite attribute comparison, $X\Theta Y$, for compatible attributes X, $Y \in \Omega - U$; Composite attribute-constant comparison, $X\Theta x$, for $X \in \Omega - U$ and $x \in \mathcal{I}_X$, and Θ is either $=, \neq, \subset, \cdots$.*

- *Let $X \in \Omega - U$. The unnesting $\mu_X(\Omega, \omega)$ equals (Ω', ω') where $\Omega' = (\Omega - \{X\}) \cup X$ and $\omega' = \{t \in \mathcal{T}_{\Omega'} | \exists t' \in \omega : t$ restricted to $\Omega - \{X\}$ equals t' restricted to $\Omega - \{X\}$ & t restricted to X is an element of $t'(X)\}$.*

- *Let $X \subseteq \Omega$. The nesting $\nu_X(\Omega, \omega)$ equals (Ω', ω') where $\Omega' = (\Omega - X) \cup \{X\}$ and $\omega' = \{t \in \mathcal{T}_{\Omega'} | \exists t' \in \omega : t$ restricted to $\Omega - X = t'$ restricted to $\Omega - X$ & $t[X] = \{t''$ restricted to $X | t'' \in \omega$ & t' restricted to $\Omega - X = t''$ restricted to $\Omega - X\}\}$.*

Example 2 Since the nest, ν, and unnest, μ, are the only different operators, we will briefly explain their effect for the relations shown in Figure 1. $R_b(A^*BC) = \nu_A(R_a)$ and

$R_c(A^*B^*C) = \nu_B(R_b)$. Note that as defined, the unnest operator is simply the inverse of the nest. As well, it is important to note that two nest operators do not necessarily commute. $\square$

The *nested algebra* $\mathcal{N}$ is defined by expressions built from typed relation variables and constant relations using the operators above. The *standard relational algebra* is the subset of $\mathcal{N}$ built with standard relation variables and standard relation constants not using nest.

An interesting approach is to allow the usage of the nest operator and the selection operator with predicates involving composite attributes internally in a query but to demand that only the standard relation variables and constants are used and that the result of the query also be standard. Paredaens and Van Gucht, [22], call these nested algebra expressions *flat-flat*, denoted ff-expressions. They also prove the following theorem which allows us to consider ff-expressions in the standard relational algebra.

Theorem 1 *For every ff-expression in the nested algebra, there is an equivalent expression in the standard relational algebra.*

2.5 A calculus for nested relations

The calculus uses typed variables ranging over tuples. The *terms* are constants, variables, and expressions of the form $x.Z$, where x is a tuple variable and $Z \in \mathcal{U}$. The *atomic formulas* are (well-typed) expressions of the form $t_1 = t_2$, $t_1 \in t_2$, or $R(t_1)$ where t_1, t_2 are terms and R is a typed relation name. Formulas are built using connectives and quantifiers in the usual manner. A *calculus query* from a database schema $\Delta = [R_1 : \Omega_1, \ldots, R_n : \Omega_n]$ to a relation scheme Ω (see footnote 1) is an expression $\{y|\Phi\}$ where Φ is built from the relation names $R_1, \ldots, R_n$ in Δ, has only y as a free tuple variable, and y has the type Ω.

Clearly, a calculus query defines a generic mapping with domain $\mathcal{D}_\Delta$. However, this mapping need not be domain preserving. We therefore also consider the notion of *domain preserving* calculus queries, and we will call the *nested calculus* the set of domain preserving calculus queries. As such the nested calculus is far more expressive than the relational algebra. We will therefore restrict our attention to the subset of the *strictly safe* queries introduced in [1, 24].

2.6 Nested SQL queries

SQL is an easy to use language which has become a standard in many standard relational database systems. The basic structure of an SQL query is of the form SELECT target_list FROM relation_list WHERE predicate. In [4], nested SQL is defined having:

- the nesting of several query blocks
- the use of keywords such as : EXISTS, ALL, ANY, IN in the connection between query blocks
- the use of variables and of predicates connecting query blocks indirectly
- the use of predicates as defined for relational queries, including set comparison
- the grouping of tuples of a relation into subsets and the evaluation of aggregate functions on each subset.

In order to minimize the confusion which could arise from the use of the word 'nested', we will refer to nested SQL as simply SQL in the remainder of the paper.

3 Nonmonotonic Queries

In this section, we have two goals. First, we will illustrate how set construction and set comparison operators are used in the formulation of nonmonotonic queries. Second, we will show how the nested algebra can be used to exhibit this use explicitly. Remember that the nonmonotonic queries have the property that when applied to a database instance which has been increased by adding tuples they may return a smaller relation. Combining results from [17, 2], the nonmonotonic queries have been identified in the following theorem.

Theorem 2 *A query containing any of the following is nonmonotonic:*

- *the relational algebra operators nest, difference, division;*
- *certain of the relational calculus queries which contain the universal quantifier or negation;*
- *any SQL query making use of the equivalent terms to those above.*

A number of approaches have been suggested to enable a database system to handle nonmonotonic queries. We discuss these approaches in the following subsections by the query language.

3.1 Relational Calculus

Pirotte, [23], and Ozsoyoglu and Wang, [21], describe how the universal quantifier and negation can be handled by extending relational calculus. In particular, in the second approach, [21] gives an algorithm which translates a relational calculus query into one in which the universal quantifiers are replaced by set comparisons and set manipulation operators and any negation is immediately to the left of an atomic formula. The subsequent language, denoted RC/C, is implemented without negation in the Statistical Database Management System, described in [20]. We exhibit this result in the following example.

Example 3 This example is drawn from [21] directly. Consider a department store database with the following relation schemes : DEPT(dname,manager,floor#), ITEM(iname, color,price,itype) and SELL(dname,item).

If we wish to find the items sold by all departments on the second floor, the resultant query would be

$$\{t^{(1)} | (\exists i)(ITEM(i) \wedge t[1] = i[1] \wedge S_P \subseteq S_Q)\}$$

where S_P and S_Q can be represented as

$$S_P = \{d | DEPT(d) \wedge d[4] =' 2'\} \textit{ and}$$

$$S_Q = \{d | (SELL(s) \wedge DEPT(d) \wedge s[2] = i[1] \wedge d[1] = s[1])\}$$

This query is nonmonotonic because if one adds a department on the second floor, the list of items sold by all departments will be constrained by the items sold in the new department. In this instance, set construction is exhibited by the definitions of the sets S_P and S_Q. Notice how S_Q depends on the existential variable i. Set comparison, in the form $\subseteq$, is between S_P and S_Q. □

3.2 SQL

SQL has been studied extensively in terms of the expressibility of nonmonotonic queries, [4, 9, 12, 18, 30]. Ceri and Gottlob, [4], provide a complete translation algorithm for SQL. We provide the following example to exhibit this.

Example 4 This example is adapted from Date, [7]. Consider a relational database with the following schemes: SUPPLIER(s#,sname), SUPPLY(p#,s#), where we wish to find the name of suppliers who do not supply product 'P2'. This query would be expressed as

```
SELECT sname
FROM   SUPPLIER
WHERE 'P2' NOT IN (SELECT p#
                   FROM   SUPPLY
                   WHERE s# = SUPPLIER.s#)
```

This query is nonmonotonic because if a supplier suddenly begins supplying part 'P2', this supplier will be dropped from the prior list of suppliers not supplying part 'P2'. In this case, set construction is exhibited by the complete nested SELECT and the set comparison by the NOT IN predicate. □

3.3 Standard Relational Algebra

The standard relational algebra includes the nonmonotonic operators difference and division. For completeness, we will exhibit the semantics of division in terms of set construction and

A	B
a_1	b_1
a_1	b_2
a_1	b_3
a_2	b_1
a_2	b_3
a_3	b_1
a_3	b_2

$R(A, B)$

B
b_1
b_2

$S(B)$

A
a_1
a_3

$R \div S$

Figure 2: Semantics of Division

set comparison. Consider the relations over $R : (A, B)$, $S : (B)$ and their division, $R \div S$ shown in Figure 2. The value a_1 (a_3) occurs in this division because the set *constructed* from the B-values associated with a_1 (a_3) *contains* the relation S. Codd, [5], and Maier, [17], show that division can be used in the translation of the universal quantifier. Dadashzadeh, [6], also describes the improved division operator which is perhaps a clearer example of set construction and set comparison. The following example shows division's use.

Example 5 The following is adapted from Graefe, [10]. Consider the following university database with two schemes: COURSES(c#) and ENROL(stud#,c#), where we wish to find the students who have taken all of the courses. This query can be simply expressed as

$$ENROL \div COURSES$$

This query is nonmontone because if a course is added to the list of courses offered, not all of the students qualifying before will have taken this new course. □

3.4 Nested Relational Algebra

The nested relational algebra directly provides both set construction and set comparison with the nest and selection over composite attributes, in addition to the standard set operators. Since SQL, standard relational algebra and relational calculus are all equivalent to the flat-flat nested relational algebra (by Theorem 2), we know that queries of these languages can be expressed directly using nested algebraic operators. We show the equivalent (flat-flat) nested relational algebra queries for each of those in examples 3, 4 and 5 below.

Example 6 For example 3, we wanted to find the items sold by all departments on the second floor of a store. The equivalent nested relational algebra query would be

$$S_P = \sigma_{floor\#='2'}(DEPT)$$

$$\Pi_{iname}(ITEM \bowtie \sigma_{S_P \subseteq dname^*}(\nu_{dname}(ITEM)))$$

For example 4, we wish the name of suppliers who do not supply part 'P2'. In nested relational algebra we would have

$$\Pi_{sname}(SUPPLIER \bowtie \sigma_{'P2' \notin p\#^*}(\nu_{p\#}(SUPPLY)))$$

Finally, for example 5 where we wish to find the students who have taken all of the courses, we would have

$$\Pi_{stud\#}(\sigma_{COURSES=ENROL.c\#^*}(\nu_{ENROL.c\#}(ENROL))) \square$$

In each of the cases, the actual translation is straightforward. The nested algebra would provide a mechanism for determining the optimization of nonmonotonic queries if the internal sets so generated can be efficiently implemented.

In addition to the single set queries listed above, there are cases when double nesting or set joins would be useful.

Example 7 Consider the store database defined in example 3. If we wish to find the pairs of departments (d_1, d_2) such that d_1 sells all of the items sold by department d_2. In nested algebra, the query would be

$$\Pi_{dname,dname'}(\nu_{iname}(SELL) \bowtie_{iname'^* \subseteq iname^*}$$

$$\nu_{iname'}(\rho_{iname \rightarrow iname'}\rho_{dname \rightarrow dname'}(SELL))) \square$$

4 Implementation of Set Operations

In order for a standard relational database to handle the internal sets identified in the previous Section, several basic conditions must be met. First, the database must have a technique for identifying sets. Next, there must be an algorithm for identifying a total order on sets.

4.1 Relational Representation of Sets

Havez and Ozsoyoglu, [11], provide an overview of techniques that can be used to identify sets, mostly in terms of the storage structures useful for nested relations and complex objects. Kim et al., [13], and Lorie et al., [16], discuss directly the issue of implementing sets within relational systems. In addition, Kuper and Vardi, [15], provide a technique for representing sets using the concepts of a *data space* and *address space*, which is applicable.

Essentially, we will discuss two distinct, but related, techniques which can be used to handle the identification of sets within standard relational databases. The first, which we call the *tag approach*, assumes that the schema of a relation is increased by adding attributes, which are used to identify the sets. The second approach, which we call the *pointer approach*, is a variation of the Kuper-Vardi approach, [15]. Since the approaches are related, we will show how they work on the following example.

Consider the relation R(A,B,C) in Figure 1(a), shown in Section 2. In Figure 1(b) and 1(c), we show the result of $\nu_A(R)$ and $\nu_B\nu_A(R)$. In the *tag approach*, we would create an attribute, denoted T_1 in Figure 3 for the first nest and attribute T_2 for the second. The values of T_1 are determined by the corresponding elements (tuples) of $\nu_A(R)$, while the values of T_2 are determined by the corresponding elements (tuples) of $\nu_B\nu_A(R)$. For example, tag t_2 is attached to the two tuples that form the second element in $\nu_A(R)$, and tag t_4 is attached to the four tuples that form the first element in $\nu_B\nu_A(R)$.

Obviously, to accomodata this technique, there must be a process outside of the relational query language which creates these tag values (or the pointers, required in the second approach), because of the explicit semantics attached to such a value. Of course, such techniques already are used in the internal processes of relational DBMS to perform say GROUP-BY in SQL.

A	B	C	T_1	T_2
1	1	1	t_1	t_4
2	1	1	t_1	t_4
1	2	1	t_2	t_4
2	2	1	t_2	t_4
3	3	2	t_3	t_5
4	3	2	t_3	t_5

Figure 3: Tag Approach

P_1	B	C
p_1	1	1
p_1	2	1
p_2	3	2

P_1	A
p_1	1
p_1	2
p_2	3
p_2	4

B	P_2
1	p_3
2	p_3
3	p_4

P_1	P_2	C
p_1	p_3	1
p_2	p_4	2

Figure 4: Pointer Approach

The *pointer approach* is exhibited in Figure 4. In this approach, extra relations are created to represent the set membership relationship for the newly created sets. Each such set is given a distinct pointer. Note that the P_1 value p_2 indicates that 3 and 4 are in the same set, and that the P_2 value p_4 indicates that 3 forms a singleton set. The pointer attributes can be used to recreate (unnest) the original relation using joins.

It should be noted that either the tag approach or the pointer approach can be used for the storage of any set valued relations, whether created through an explicit ν or through other set constructors (union, difference, etc).

4.2 Complexity

Having shown how the temporary nested relations could be implemented in a relational DBMS, we now turn our attention to the problem of how to implement the set algebra operations. It should be clear, as noted in [8, 10, 26], that the first nest operation could be implemented as a simple sort on the attributes not mentioned in the operation. However, a second and later nests require us to partition the data based on equality over set as well as elementary attributes. This same observation applies to all of the other (extended) algebra expressions. For example, duplicate elimination in a projection requires the comparison of set-valued attributes. In order to achieve this, we require a definition of *ordering* of sets that extends easily to tuples containing sets as values. Many such orderings exist, but we have selected a total ordering which preserves set containment.

We assume that there is a total order, $<$, on the values of the elementary attributes. Clearly this order can be naturally extended to tuples consisting of elementary values. We that when X is a composite attribute made up of elementary attributes, and S_1 and S_2 $\in \mathcal{I}_X$, $S_1 \prec S_2$, if $|S_1| < |S_2|$, or $|S_1| = |S_2|$ and for the least element different in S_1 and S_2, $(s \in S_1, t \in S_2)$, $s < t$.

Example 8 For the simple sets over the domain $\{0,1,2\}$, we have $\emptyset \prec \{0\} \prec \{1\} \prec \{2\} \prec \{0,1\} \prec \{0,2\} \prec \{1,2\} \prec \{0,1,2\}$.

For the simple sets over two attributes from domain $\{0,1,2\}$, we have $\emptyset \prec \{[00]\} \prec \{[01]\} \prec \{[02]\} \prec \{[10]\} \prec \cdots \prec \{[22]\} \prec \{[00],[01]\} \prec \{[00],[02]\} \prec \cdots \prec \{[20],[21],[22]\}$.
$\square$

We can now show that this ordering can be reasonably efficiently computed. We will show the result for the tag approach of representing sets, even though the pointer approach will be more efficient on average. In the worst case, the complexity of both approaches will be the same.

Lemma 1 *A bag*[2] *of sets of tuples of elementary attribute values, $C = \{S_1, S_2, \ldots, S_k\}$ can be ordered according to $\prec$ in time proportional to $O(n \log n)$, where n is the number of tuples necessary to represent the bag in the tag approach.*

Proof: We first note that a set S of tuples of elementary attributes can be sorted using an $O(|S| \log |S|)$ algorithm, say a merge-sort. This of course assumes that the number of elementary attributes is fixed. Secondly, we note that the number of tuples, n, necessary

[2] A bag is a set in which certain elements may be repeated.

to represent the bag C is $\sum_{i=1}^{k} |S_i|$, since each S_i will contribute $|S_i|$ tuples to the tag representation of C. The first step in our algorithm to order C is to sort each set on the elementary attributes. As part of this step, we also record the number of tuples in each set, which can be done in one extra pass through the tuples. In total, this step would require $O(\sum_{i=1}^{k} |S_i| log|S_i|) \leq O(\sum_{i=1}^{k} |S_i| log\, n) \leq O(n\, log\, n)$, since $n = \sum_{i=1}^{k} |S_i|$. Since the sets, S_i, are now in tuple order, a merge-sort can be defined for $\prec$ as follows. For each pair of sets, (S_i, S_j), in a merge-sort pass, compare the number of elements in S_i and S_j first. If one set contains more elements than another, that set is larger according to $\prec$. If not, then because S_i and S_j have been previously sorted, each set can be compared tuple by tuple (first tuple in S_i against the first tuple in the S_j, etc). Since each tuple in a set will contribute exactly one tuple to the tag representation, to compare two sets would require comparing at most the number of tuples representing the sets in the tag approach. Thus, one pass of the merge-sort would require at most the number tuples necessary to represent the whole bag, $O(n)$. Since at most $log\, k$ passes are required in a merge-sort and $k \leq n$, we can order the bag in $O(n\, log\, n)$ steps. $\square$

It is interesting to note that computing the number of tuples in each set is required in this algorithm and will be available for aggregate functions immediately.

The order $\prec$ can now be extended recursively to tuples whose domains can be elementary attributes, or composite attributes at any depth of nesting naturally. We will assume for ease of discussion that the attributes are ordered from elementary to most complex.

Example 9 Consider a collection of tuples defined over a schema $R(A(BC^*)^*)$, where A, B and C are over domain $\{0,1\}$. Then we would have the following order : $[\,] \prec [0\emptyset] \prec [0\{0\emptyset\}] \prec [0\{0\{0\}\}] \prec [0\{0\{1\}\}] \prec [0\{0\{0,1\}\}] \prec [0\{1\emptyset\}] \prec \cdots \prec [1\emptyset] \prec \cdots \prec [1\{1\{0,1\}\}]$. $\square$

This ordering as well can be efficiently achieved as shown below.

Lemma 2 *Ordering a bag of tuples $C = \{S_1, S_2, \ldots, S_m\}$ can be achieved in time propor-*

tional to $O(n \log n)$, where n is the number of tuples required to represent C in the tag approach.

Proof Consider the tuples having the attributes as $X_1, X_2, \ldots, X_l$, where the attributes are ordered from elementary to most complex composite attribute. Each X_i is considered separately. If X_i is a composite attribute, denote as p_{ij} the number of tuples to represent $S_j[X_i]$ in the tag approach. Then, from the tag approach we know that $\sum_{j=1}^{m} p_{ij} \leq n$. Since the iterative application of the sort-merge would require sorts on each of the size p_{ij} sets, we arrive at $\sum_{j=1}^{m} p_{ij} \log p_{ij} \leq \sum_{j=1}^{m} p_{ij} \log n \leq n \log n$. Since each of the attributes is considered only if all of the previous attribute values are equal, at most l applications of the ordering are required. Thus, the whole ordering can be done in $O(n \log n)$ time. □

The following theorems follows immediately.

Theorem 3 *An instance of a nested relation can be ordered according to $\prec$ in time proportional to $O(n \log n)$, where n is the number of tuples required to represent the instance in the tag approach.*

Theorem 4 *Any of the set constructor and set comparison operators can be achieved in time proportional to $O(n \log n)$, where n is the number of tuples required to represent the data in the tag approach.*

Proof Since the ordering $\prec$ determines set containment directly, only one pass through the ordered tables is required to determine it. For ν, we need only to pass through the table creating a new value for a created attribute when the adjacent sets are not equal. Comparisons between attributes can be done in the same way. The basic set operators union and difference by ordering the two relation instances in the same order and making a single pass through each table, which at worst doubles the size of n. □

The only seemingly difficult operations are cross-product and join. However, the problem with these operators is that the size of the output after applying them can grow as a product of the sizes of the input databases. Since the ordering algorithm given above will order each of the input databases in $O(n\,log\,n)$ steps, for n the size of the larger, to compute them will be bound by the size of their output in the tag representation. The final theorem follows immediately.

Theorem 5 *For any ff-query on a relational database, the total time required to compute a query is bound by a polynomial whose power is determined by the number of cross products required in the query evaluation.*

This theorem, in combination with Theorem 1, yields the following fact.

Fact 1 *The explicit usage of set construction and set comparison operators is*

- *a safe and often succinct tool for the formulation of nonmonotonic queries,* and
- *can be efficiently supported in existing relational DBMS, i.e., the efficiency is no worse than the efficiency associated with the implementation of query languages without explicit set construction and set comparison.*

In order to show the efficiency of the approach, consider the following example.

Example 10 Consider a relation $R(A, B, C)$ and the query $\pi_C(\sigma_{A^* \subseteq B^*}(\nu_B \nu_A R))$ applied to the values in Figure 1. The query would return 1 and not 2. By Theorem 4, each of ν_B and ν_A can be calculated in $O(n\,log\,n)$ steps, where $n = |R|$, even though one nest is at the internal level. The σ operator will require at most $O(n)$ steps, since the elements are already in order. Similarly, the π_C will require only $O(n)$ steps. In total, only $O(n\,log\,n)$ are required. □

5 Conclusion

We have shown that the nonmonotonic queries can be considered consistently as set construction and set comparison operators and that the nested relational algebra provides a natural language in which to implement these set operators. Finally, we showed that the operators can be efficiently implemented within a standard relational management system internally without adding to the power of the system. Thus, the nested algebra can be used in an optimization routine for any standard relational database management system for the nonmonotonic queries.

Several outstanding research problems are evident. For the relational storage of sets, we need to determine the most space and time efficient method for specific commercial systems. As well, given that most commercial systems provide indexing schemes, for example B-trees, and hashing, the most efficient way of actually ordering sets is open, Kim et al., [13], and Graefe, [10], provide guidelines to this study. This leads as well to further investigations as to the complete implementation of a nested database as a base for an object-oriented system, [13, 25, 29].

References

[1] S. Abiteboul and C. Beeri. On the power of languages for the manipulation of complex objects. Technical report, INRIA, 1988.

[2] C. Beeri and Y. Kornatzky. The many faces of query monotonicity. In *Proc. of Advances in Database Technology-EDBT '90*, pages 120–135, 1990.

[3] C. Beeri, S. Naqvi, O. Shmueli, and S. Tsur. Sets and negation in logic database language (LDL), rev. 1. Technical Report DB-375-86, MCC, 1987.

[4] S. Ceri and G. Gottlob. Translating SQL into relational algebra: Optimization, semantics, and equivalence of SQL queries. *IEEE Trans. on Software Engineering*, SE-11(4):324–345, April 1985.

[5] E.F. Codd. A relational model for large shared data banks. *Comm. of the ACM*, 13:377–387, 1970.

[6] M. Dadashzadeh. An improved division operator for relational algebra. *Information Systems*, 14(5):431–437, 1989.

[7] C.J. Date. *An Introduction to Database Systems*, volume 1. Addison-Wesley, Reading, MA, 5th edition, 1990.

[8] U. Dayal. Of nests and trees: A unified approach to processing queries that contain nested subqueries, aggregates, and quantifiers. In *Proc. of 13th VLDB Conf.*, pages 197–208, 1987.

[9] R. Ganski and H. Wong. Optimization of nested SQL queries revisited. In *Proc. 1987 ACM SIGMOD Int'l Conf. on Management of Data*, pages 23–33, 1987.

[10] G. Graefe. Relational division: Four algorithms and their performance. In *Proc. 1989 IEEE Conf. on Data Engineering*, pages 94–102, 1989.

[11] A. Hafez and G. Ozsoyoglu. Storage structures for nested relations. *IEEE Data Engineering*, pages 31–28, 1988.

[12] W. Kim. On optimizing an SQL-like nested query. *ACM Trans. on Database Systems*, 7(3):443–469, 1982.

[13] W. Kim, H.-T. Chou, and J. Banerjee. Operations and implementation of complex objects. *IEEE Trans. on Software Engineering*, 14(7):985–996, 1988.

[14] A. Klug. Equivalence of relational algebra and relational calculus query languages having aggregate functions. *JACM*, 29(3):699–717, 1982.

[15] G.M. Kuper and M.Y. Vardi. A new approach to database logic. In *Proc. 1984 ACM SIGACT-SIGMOD PODS Conf.*, pages 86–96, 1984.

[16] R. Lorie, W. Kim, D. McNabb, W. Plouffe, and A. Meier. Supporting complex objects in a relational system for engineering databases. In W. Kim, D. Reiner, and D. Batory, editors, *Query Processing in Database Systems.* Springer-Verlag, 1985.

[17] D. Maier. *The Theory of Relational Databases.* Computer Science Press, Rockville, MD, 1983.

[18] M. Muralikrishna. Optimization and dataflow algorithms for nested tree queries. In *Proc. of 15th VLDB Conf,*, pages 77–85, 1989.

[19] G. Ozsoyoglu and V. Matos. On optimizing summary-table-by-example queries. In *Proc. 1985 ACM SIGACT-SIGMOD PODS Conf.*, pages 38–49, 1985.

[20] G. Ozsoyoglu, V. Matos, and Z.M. Ozsoyoglu. Query processing techniques in the summary-table-by-example database query language. *ACM Trans. on Database Systems*, 14(4):526–573, 1989.

[21] G. Ozsoyoglu and H. Wong. A relational calculus with set operators, its safety, and equivalent graphical languages. *IEEE Trans. on Software Engineering*, 15(9):1038–1052, 1989.

[22] J. Paredaens and D. Van Gucht. Converting nested algebra into flat relational structures. to appear *ACM Trans. on Database Systems.*

[23] A. Pirotte. High level data base query languages. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 409–436. Plenum Press, New York, NY, 1978.

[24] M.A. Roth, H.F. Korth, and A. Silberschatz. Extended algebra and calculus for nested relational databases. *ACM Trans. on Database Systems*, 13(4):389–417, 1988.

[25] L.A. Rowe and M. Stonebraker. The POSTGRES data model. In *Proc. 13th VLDB*, pages 83–96, 1987.

[26] G. Saake, V. Linnemann, P. Pistor, and L. Wegner. Sorting, grouping and duplicate elimination in the advanced information management prototype. In *Proc. of 15th VLDB Conf.*, pages 307–316, 1989.

[27] M.H. Scholl. Theoretical foundation of algebraic optimization utilizing unnormalized relations. In *Proc. Int'l Conf. on Database Theory*, pages 380–396, 1986.

[28] M.H. Scholl, H.-B. Paul, and H.-J. Schek. Supporting flat relations by a nested relational kernel. In *Proc. 13th VLDB Conf.*, pages 137–146, 1987.

[29] M. Stonebraker. The case for partial indexes. *ACM SIGMOD Record*, 18(4):4–11, 1989.

[30] G. von Bultzingsloewen. Translating and optimizing SQL queries having aggregates. In *Proc. of 13th VLDB Conf.*, pages 235–243, 1987.