Optimization of Nested Queries in a Complex Object Model

Hennie J. Steenhagen, Peter M.G. Apers, and Henk M. Blanken

Department of Computer Science, University of Twente,
PO Box 217, 7500 AE Enschede, The Netherlands

Abstract. Transformation of nested SQL queries into join queries is advantageous because a nested SQL query can be looked upon as a nested-loop join, which is just one of the several join implementations that may be available in a relational DBMS. In join queries, dangling (unmatched) operand tuples are lost, which causes a problem in transforming nested queries having the aggregate function COUNT between query blocks–a problem that has become well-known as the COUNT bug. In the relational context, the outerjoin has been employed to solve the COUNT bug. In complex object models supporting an SQL-like query language, transformation of nested queries into join queries is an important optimization issue as well. The COUNT bug turns out to be a special case of a general problem being revealed in a complex object model. To solve the more general problem, we introduce the *nest join* operator, which is a generalization of the outerjoin for complex objects.

1 Introduction

Currently, at the University of Twente, work is being done on the high-level object-oriented data model TM. TM is a database specification language incorporating standard object-oriented features such as classes and types, object identity, complex objects, and multiple inheritance of data, methods, and constraints. In TM, methods and constraints are specified in a high-level, declarative language of expressions. An important language construct is the SELECT-FROM-WHERE (SFW) construct. The SFW-construct of TM is comparable to the SFW-query block from HDBL, the query language of the experimental DBMS AIM [10]. HDBL is an orthogonal extension of SQL for extended NF^2 data structures. Optimization of TM SFW-expressions therefore has much in common with optimization of the SFW-expressions of SQL and HDBL.

Optimization of SQL queries has received quite some attention the last decade. An important problem in this area is the optimization of nested SQL queries [7, 5, 4, 8, 9]. SQL offers possibilities to formulate nested queries: SFW-query blocks containing other SFW-blocks in the WHERE clause. In [7], it was pointed out that it is advantageous to replace nested SQL queries by flat, or join queries. Flat SQL queries are SFW-blocks not containing subqueries in the WHERE clause. Replacing nested SQL queries by join queries is advantageous because a nested SQL query can be looked upon as a nested-loop join, which is just one of the several join implementations possible. After rewriting a nested query into a join query, the optimizer has better possibilities to choose the most appropriate join implementation.

In nested queries, inner operand tuples are grouped by the values of the outer operand tuples. Whenever aggregate functions occur between query blocks, the transformed, i.e. join query also requires grouping (expressed by means of the GROUP BY clause). In nested queries, dangling outer operand tuples, i.e. outer operand tuples that are not matched by any of the inner operand tuples, deliver a subquery result equal to the empty set. Transformation of a nested query into a join query causes the loss of dangling tuples. In the relational context, this may cause a problem when the aggregate function COUNT occurs between query blocks. As a solution to this problem (that has become well known as the COUNT bug), it has been proposed to use the outerjoin instead of the regular join [5].

In complex object models supporting an SQL-like query language, transformation of nested queries into join queries is just as important as in the relational context. However, in complex object models grouping of the inner operand is required not only if aggregate functions occur between query blocks, but in many other cases as well. The reason for this is that attributes may be set valued. Moreover, each time grouping is necessary, we have to deal with some kind of COUNT bug caused by the loss of dangling tuples, i.e. the COUNT bug is just a special case of a more general problem being revealed in a complex object model. An important result of this paper is that from the form of the predicate between query blocks it can easily be derived when grouping is not necessary. Nested queries that do not require grouping can be transformed into join queries; for the efficient and correct processing of nested queries that do require grouping a new join operator is introduced–the *nest join* operator.

Instead of producing the concatenation of every pair of matching tuples as in the regular join operation, in the nest join operation each left operand tuple is extended with the *set* of matching right operand tuples. This way two birds are killed with one stone: grouping is performed, and also dangling tuples are preserved. Implementation of the nest join operator is a simple modification of any common join implementation method, however, like the outerjoin operator, the nest join has limited rewrite possibilities compared to the regular join operator.

In general, in the logical optimization of a declarative query language, two approaches can be distinguished: (1) rewriting expressions in the query language itself and (2) translation into and rewriting in some intermediate language, for example an algebraic language. Also a combination of the two approaches is possible. For the logical optimization of TM, we have defined the language ADL, an algebra for complex objects which is an extension of the NF^2 algebra of [12]. This work will be used in the ESPRIT III project IMPRESS (Integrated, Multi-Paradigm, Reliable, and Extensible Storage System). The IMPRESS project started in 1992, and one of the subtasks is to translate (a subset of) the language TM into an algebra for complex objects resembling ADL. In this paper, our ideas with regard to query transformation will be presented using the language TM; we will not introduce the algebra for reasons of simplicity.

The structure of this paper is as follows. In Section 2, we briefly review the work that has been done with regard to nested SQL queries. In Section 3, we describe the language TM and the types of nested SFW-expressions that are of interest for the purpose of this paper. Nesting in the WHERE clause and in the SELECT clause are discussed in Sections 4 and 5, respectively, and we will see that in the transformation of nested queries into join

queries in many cases grouping is needed, each case leading to some kind of COUNT bug if relational transformation techniques are used. Then, in Section 6, we introduce the nest join operator. The nest join is an operator that allows efficient processing of nested queries that cannot be transformed into join queries without grouping. Bugs are avoided by preserving dangling tuples. In Section 7 we show, for two-block queries, which types of nested queries can be transformed into join queries without problem. An example of query processing for an arbitrary linear nested query (having only one subquery per WHERE clause) is then given in Section 8. The paper is concluded by a section discussing future work.

2 Nested SQL Queries

In this section we briefly review the work that has been done on optimization of nested SQL queries. We do not give a complete overview; we merely indicate the ideas behind optimization of nested SQL queries with a view on the additional problems that come up with optimization of nested queries in a data model supporting complex objects.

Nested SQL queries are SFW-query blocks containing other (possibly nested) SFW-query blocks in the WHERE clause. For example, assume we have relation schemas $R(A, B, C)$ and $S(C, D)$, and consider the following SQL query:

SELECT *
FROM R
WHERE $R.B$ IN SELECT $S.D$
FROM S
WHERE $R.C = S.C$

Disregarding duplicates, the nested query given above is easily transformed into the flat, or join query:

SELECT $R.A$, $R.B$, $R.C$
FROM R, S
WHERE $R.B = S.D$ AND $R.C = S.C$

which, in relational algebra, is simply a join between tables R and S followed by a projection on R, i.e. a semijoin. The advantage of transforming nested queries to join queries is clear: a nested-loop join is just one of the several possible implementations of the join operator, and after transformation to a join query the optimizer can choose the most suitable join execution method. The method chosen may be a nested-loop join, but not necessarily–alternative join implementations are the sort-merge join, the hash join etc.

In [7], five types of nesting have been distinguished and an algorithm has been given to transform nested queries into join queries for each of these different types of nesting. In case aggregate functions occur between query blocks (one of the types of nesting) SQL's GROUP BY clause is employed to compute the aggregates needed. However, in [6] it has been shown that Kim's algorithm is not correct if the aggregate function COUNT occurs between query blocks. This flaw has become known as the COUNT bug. Consider the query:

SELECT *
FROM R
WHERE $R.B$ = SELECT COUNT (*)
FROM S
WHERE $R.C = S.C$

Following Kim's algorithm, we get the following queries:

(1) $T(C, CNT)$ = SELECT $S.C$, COUNT (*)
FROM S
GROUP BY $S.C$

SELECT $R.A, R.B, R.C$
FROM R, T
WHERE $R.B = T.CNT$ AND $R.C = T.C$

Alternatively, if the relation R does not contain duplicates, the nested query may be transformed into:

(2) SELECT $R.A, R.B, R.C$
FROM R, S
WHERE $R.C = S.C$
GROUP BY $R.A, R.B, R.C$
HAVING $R.B$ = COUNT ($S.C$)

In the former, grouping of the inner operand and computation of the aggregate precedes the join operation; in the latter the join is executed first.

The queries resulting from the transformations do not give the correct result. In the original, nested query, dangling R-tuples for which $R.B = 0$, are included in the result; these tuples are lost in the join queries.

To solve the COUNT bug, it has been proposed in [5] to modify (2) by using outerjoins instead of joins in case the COUNT function occurs between query blocks. The right outerjoin operator preserves dangling tuples of the left join operand: unmatched left operand tuples are extended with NULL values in the right operand attribute positions.

As another solution, it has been proposed in [9] to modify (1), because in some cases (1) is more efficient than (2). It is proposed to have two types of join predicates in the second query of (1): a regular join predicate and an additional, so-called antijoin predicate, to be applied to the dangling tuples. In the example given above the antijoin predicate would be: $R.B = 0$. In Kim's second query the join is replaced by an outerjoin operation with join predicate $R.C = T.C$; to the tuples that match the predicate $R.B = T.CNT$ is applied, and to the unmatched tuples in R the antijoin predicate $R.B = 0$ is applied.

3 Nested TM Queries

3.1 General Description of TM

In this section we describe the features of TM that are important for the purpose of this paper–support for complex objects and the SELECT-FROM-WHERE construct. For a more comprehensive description of TM we refer the reader to [1, 2, 3].

TM is a high-level, object-oriented database specification language. It is formally founded in the language FM, a typed lambda calculus allowing for subtyping and multiple inheritance. Characteristic features of TM are the distinction between types, classes, and sorts, support for object identity and complex objects, and multiple inheritance of attributes, methods, and constraints. In TM, attribute types may be arbitrarily complex: the type constructors supported are the tuple, variant, set, and list type constructor; type constructors may be arbitrarily nested. Besides basic types, class names may be used in type specifications. Sets do not contain duplicates.

In constraint and method specifications we may use the SELECT-FROM-WHERE construct, having the following general format:

SELECT <result expression>
FROM <operand expression> <variable>
WHERE <predicate>

The meaning of the SFW-expression is as follows. The operand expression is evaluated; a variable is iterated over the resulting set; for each value of the variable it is determined whether the predicate holds, and if so, the result expression is evaluated and this value is included in the resulting set.

3.2 Types of Nesting in TM

One important difference between SQL on the one hand, and TM and HDBL on the other is that TM and HDBL are *orthogonal* languages. The operand and result expression of the SFW-query block of TM may be arbitrary expressions, also containing other (nested) SFW-expressions, provided they are correctly typed. The predicate may also be built up from arbitrary expressions (including quantifiers FORALL and EXISTS), as long as it delivers a Boolean result.

We give some examples of SFW-expressions. Assume we have specified classes 'Employee' and 'Department':

```
CLASS Employee WITH EXTENSION EMP
ATTRIBUTES
        name : STRING,
        address : Address,
        sal : INT,
        children : ℙ⟨name : STRING, age : INT⟩
END Employee

CLASS Department WITH EXTENSION DEPT
ATTRIBUTES
        name : STRING,
        address : Address,
        emps : ℙEmployee
END Department

SORT Address
TYPE ⟨street : STRING, nr : STRING, city : STRING⟩
END Address
```

The symbol $\mathbb{P}$ denotes the set type constructor, brackets $\langle\rangle$ denote the tuple type constructor. In TM, class extensions are explicitly named. The class 'Employee' has four attributes, of which the attribute 'address' has a complex type specified as a sort. Sorts are used to describe commonly used types such as 'Address', 'Date', 'Time' etc..

Q1: Select the departments that have at least one employee living in the same street the department is located.

```
SELECT d
FROM DEPT d
WHERE ⟨s = d.address.street, c = d.address.city⟩
        IN SELECT ⟨s = e.address.street, c = e.address.city⟩
           FROM d.emps e
```

Q2: Select for all departments the names of the departments and the employees living in the same city the department is located.

```
SELECT ⟨dname = d.name, emps = SELECT e
                               FROM EMP e
                               WHERE e.address.city = d.address.city⟩
FROM DEPT d
```

We make a distinction in the types of nested queries. In a SFW-expression, other SFW-expressions may occur in the SELECT clause (query $Q2$), in the FROM clause, and in the WHERE clause (query $Q1$). In this paper, the expressions of interest are nested SFW-expressions having subqueries in which free variables (correlated subqueries) occur; subqueries without free variables simply are constants. We do not consider SFW-expressions with subqueries in the FROM clause, because these can be rewritten easily. Furthermore, operands of subqueries may be either set-valued attributes, as in query Q1, or distinct tables, as in query Q2. Only if subquery operands are distinct tables, transformation to join queries is desirable. There is no use to flatten nested queries in which subquery operands are set-valued attributes, because set-valued attributes are stored with the objects themselves (as materialized joins), at least conceptually.

In short, the nested queries of interest are SFW-expressions having subqueries in the SELECT- and/or WHERE clause containing free variables and having distinct tables as operands. Initially, we will restrict ourselves to two-block queries. In Section 8 we will briefly discuss queries with multiple nesting levels.

4 Nesting in the WHERE Clause

Assume we have a two-block query with one-level deep nesting. The general format of such a query is:

```
SELECT F(x)
FROM X x
WHERE P(x, z)
        WITH z = SELECT G(x, y)
                 FROM Y y
                 WHERE Q(x, y)
```

The WITH clause is a TM construct enabling local definitions, used here to facilitate the description of the syntactical form of the predicate P. In this paper, we do not consider multiple subqueries, $P(x, z)$ contains only one occurrence of z.

We want to transform the nested query into a join query of the following format (remember that, in SQL, grouping is necessary only if aggregate functions occur between query blocks):

SELECT $F(x)$
FROM $X\ x, Y\ y$
WHERE $P'(x, v) \wedge Q(x, y)$
WITH $v = G(x, y)$

For notational convenience, also the expression $G(x, y)$ has been named by means of a WITH clause.

The goal of the transformation process is to transform the predicate $P(x, z)$, whose second argument z is set valued, into a predicate $P'(x, v)$, where values v are the members of z. The types of P and P' clearly differ: from the second argument of P a set constructor is removed, resulting in predicate P'.

4.1 Example Predicates

Assume that the predicate P only involves attribute a of the outer operand X and z, the subquery result. Because the attribute a may be set valued, this (already restricted) predicate between query blocks may take many different forms. We may have for instance[1]:

- $x.a\ OP\ z$ with $OP \in \{\in, \subset, \subseteq, =, \supseteq, \supset, \ni\}$,
- expressions involving quantifiers, for example $\exists s \in z\ (s = x.a)$,
- $x.a\ OP\ H(z)$ with H an aggregate function and OP an arithmetical comparison operator,
- expressions involving set operators, for example $x.a \cap z = \emptyset$, or
- negations of expressions listed above.

Predicates can be divided into two groups: predicates that require grouping of the inner operand tuples and the predicates that do not. In Section 7, we give a formal characterization of predicates that do and do not require grouping; below, the need for grouping is discussed more informally.

Grouping is not necessary if the question whether outer operand tuples belong to the result or not (whether, for some outer operand tuple, the predicate evaluates to $true$ or $false$) can be answered on the basis of the individual members of the subquery result, i.e. by *scanning* the subquery result. For example, consider the expression $x.a \in z$. The moment we encounter a tuple y in the inner operand Y such that the condition $Q(x, y)$ holds and $x.a$ equals the value of v, we know that tuple x belongs to the result. If no such v is found in the end, the predicate evaluates to $false$.

[1] In the rest of the paper we will use the common set-theoretical notation for comparison operators and boolean connectives occurring in TM-predicates because of its conciseness.

Grouping is necessary if the subquery result has to be available *as a whole* to decide whether the predicate holds. In this case, all tuples belonging to the subquery result must be kept, because the predicate can only be evaluated having all values in the subquery result at hand. An obvious example of a predicate requiring grouping is the expression $x.a = \text{COUNT}(z)$: not until the entire subquery result is at our disposal it is possible to compute (or output, if accumulated) the cardinality of the subquery result. Another predicate requiring grouping is for example the expression $x.a \subseteq z$.

Whenever a predicate needs grouping, we have to deal with some sort of COUNT bug if the nested query is transformed according to the algorithm of [7]. For example, the nested query:

SELECT x
FROM X x
WHERE $x.a \subseteq z$ WITH z = SELECT $y.a$
FROM Y y
WHERE $x.b = y.b$

is, following the ideas of [7], transformed into the following TM queries:

T = SELECT $\langle b = y.b,\ as$ = SELECT $y'.a$
FROM Y y'
WHERE $y'.b = y.b\rangle$
FROM Y y

SELECT x
FROM X x, T t
WHERE $x.b = t.b \wedge x.a \subseteq t.as$

The first query groups the set of $y.a$ values by the values of the attribute b (cf. the operator nest (ν) from the NF^2 algebra [12]). The transformed query also suffers from a bug (which we might call the SUBSETEQ bug in this case): X-tuples for which $x.a = \emptyset$ that are not matched by any t-tuple on the condition $x.b = t.b$ are lost.

In summary, in TM grouping of the inner operand is required not only if aggregate functions occur between query blocks, but in many other cases too. If Kim's solution is chosen, the transformed query will suffer from a bug each time grouping is needed. We will not use the outerjoin operator to solve these bugs. Instead, in Section 6, we will introduce the nest join operator, which is a much cleaner solution in a model supporting complex objects.

5 Nesting in the SELECT Clause

In this section, we will show that, with one notable exception, nesting in the SELECT clause always requires grouping of the inner operand. SFW-expressions having subqueries in the SELECT clause are not new. In HDBL, it is also allowed to have SFW-query blocks in the SELECT clause. SFW-expressions nested in the SELECT clause commonly describe nested results, as in query Q2 from Section 3.2 where employees are grouped by departments. Consider the general format of a two-block query with nesting in the SELECT clause:

SELECT $F(x, z)$
WITH z = SELECT $G(x, y)$
FROM Y y
WHERE $Q(x, y)$
FROM X x
WHERE $P(x)$

If this query is to be transformed into a join query, the inner operand values have to be grouped by the outer operand values. Grouping may take place preceding or following the join. In both cases again dangling tuples are lost.

With regard to nesting in the SELECT clause, there is one special case in which grouping can be avoided. In TM, a SFW-expression may be nested directly in the SELECT clause, meaning the result is a set of sets. This set of sets may be 'collapsed' by applying the operator UNNEST, which is defined as UNNEST$(S) = \bigcup\{s \mid s \in S\}$. Consider the following query:

UNNEST (SELECT (SELECT $\langle a = x.a, b = y.b \rangle$
FROM Y y
WHERE $x.b = y.a$)
FROM X x

This nested query is equivalent to the join query:

SELECT $\langle a = x.a, b = y.b \rangle$
FROM X x, Y y
WHERE $x.b = y.a$

6 The Nest Join Operator

In the previous sections we have shown that in a complex object model, in many cases grouping seems to be an essential step in the transformation of nested queries to join queries. Queries requiring grouping may always be handled by means of nested-loop processing, which gives correct results but may be very inefficient. If we, though, choose to transform nested queries into join queries, we have to take special measures when queries need grouping because dangling tuples are lost. In the relational model the outerjoin is used to take care of dangling tuples: for subqueries that deliver empty sets the NULL is used to represent the empty set. In a complex object model, however, we do not have to represent the empty set: *the empty set is part of the model.*

Definition

The nest join operator, denoted by the symbol $\triangle$, is simply a modification of the join operator. Instead of producing the concatenation of every pair of matching tuples, for each left operand tuple a set is created to hold the (possibly modified) right operand tuples that match. The nest join of two tables X and Y on predicate Q with function G (the function applied to the right-hand tuples satisfying the join predicate) is defined as:

$$X \underset{x,y:Q,G;a}{\triangle} Y \stackrel{d}{=} \{x ++ \langle a = z \rangle \mid x \in X \wedge z = \{G(x,y) \mid y \in Y \wedge Q(x,y)\}\}$$

In this expression, $x ++ \langle a = z \rangle$ denotes the concatenation of the tuple x and the unary tuple $\langle a = z \rangle$, in which a is an arbitrary label not occurring on the top level of X. An example of the nest join operation is found in Table 1, where flat relations X and Y are equijoined on the second attribute (the join function is the identity function). Note that for dangling tuples $x \in X$, the tuple $x ++ \langle a = \emptyset \rangle$ is present in the result. The nest

a	b
1	1
1	2
2	3

c	d
1	1
2	1
3	3

a	b	$s(c,d)$
1	1	{(1,1), (2,1)}
1	2	$\emptyset$
2	3	{(3,3)}

Table 1. X, Y, and the nest equijoin of X and Y on the second attribute

join operation is a neatly defined operation. Grouping, which is performed during the join, is made explicit by means of a set-valued attribute. Because dangling tuples are preserved, bugs like the COUNT bug are prevented.

Algebraic Properties

Assuming the nest join function is identity, the nest join can be expressed using the outerjoin, denoted by the symbol $\odot$, and the nest operator ν^*:

$$X \underset{x,y:Q,id;a}{\triangle} Y \equiv \nu^*_{X;a}(X \underset{x,y:Q}{\odot} Y)$$

In this algebraic expression, the operator ν^* is a slightly modified version of the standard nest operator performing nesting in the usual way, and then mapping nested sets consisting of a NULL-tuple to the empty set [13]. By using the nest join instead of the outerjoin followed by the nest operator, we do not have to resort to NULLs to avoid the loss of dangling tuples.

A disadvantage of the nest join operator is that it, like the outerjoin, has less pleasant algebraic properties. For example, the nest join operation is neither commutative nor associative. As another example, the nest join does not always associate with the regular join: $X \triangle (Y \bowtie Z)$ is not equivalent to $(X \triangle Y) \bowtie Z$, the two expressions already being typed differently. Below we list some examples of algebraic equivalences concerning the nest join operator.

Let $X \triangle_p Y$ denote a nest join operation on predicate p in which the nest join function equals the identity function (for simplicity omitting variable names and the nest join label). Let $r(a,b)$ denote a predicate referencing attributes in tables A and B (and no other), then we have:

- $\pi_X(X \triangle Y) \equiv X$
- $(X \bowtie_{r(x,y)} Y) \triangle_{r(x,z)} Z \equiv (X \triangle_{r(x,z)} Z) \bowtie_{r(x,y)} Y)$
- $(X \bowtie_{r(x,y)} Y) \triangle_{r(y,z)} Z \equiv X \bowtie_{r(x,y)} (Y \triangle_{r(y,z)} Z)$

Implementation

To implement the nest join, common join implementation methods like the sort-merge join, or the hash join can be used. However, some restrictions hold. First, in nest join implementations, an output tuple can be produced not until the entire set of matching right operand tuples has been found. Second, because in the nest join operation the output has to be grouped according to the values of the left operand tuples, the choice for outer and inner loop operand is restricted. For example, in a (simple) hash join implementation, if the join attribute does not form a key attribute of the right join operand, only the right join operand may be the build table. (For the regular join, usually the smaller operand is chosen as the build table.)

Use

The nest join operation can be employed to process queries with nesting in the SELECT- as well in the WHERE clause. Queries having subqueries in the SELECT clause often describe nested results, so processing by means of the nest join operation will be an appropriate method of processing. For queries with nesting in the WHERE clause, however, sometimes there are other, more efficient possibilities. In the following section we show in which cases grouping certainly is not necessary, so that, instead of the nest join operator, we may employ some flat join operator to obtain the results needed.

7 The Need for Grouping

In this section, we present a class of predicate expressions for which it is known that grouping is *not* necessary. Again consider the general format of a two-block query with nesting in the WHERE clause given in Section 4, then we have:

Theorem 1. *Grouping is* not *necessary if the predicate expression* $P(x, z)$ *can be rewritten into a calculus expression of the form (1)* $\exists v \in z\ (P'(x, v))$ *or (2)* $\nexists v \in z\ (P'(x, v))$. *In this expression,* $P'(x, v)$ *may be arbitrary.*

Proof of Theorem 1 is omitted due to lack of space. It is an open question whether grouping is always necessary in case predicate P cannot be rewritten into one of the two forms given above.

Generally, a nested query may be processed by applying nest join operators, possibly followed by (nested) function applications (e.g. projections) and selections. However, sometimes nest join operators may be replaced by flat join operators. For a two-block query, in case the predicate between the two blocks can be rewritten into a calculus expression of the first format, involving a non-negated existential quantifier, a semijoin operation will provide the correct result. If it is possible to rewrite the predicate into a calculus expression involving a negated existential quantifier, then the flat join operation needed is the antijoin operation. The join predicate is $P'(x, G(x, y)) \wedge Q(x, y)$. (Remember that the semi-and antijoin operations are defined as follows. Let X and Y be tables (sets of tuples having possibly complex attributes), and let P be a predicate, then the semijoin operation $X \ltimes_{x,y:P} Y$ is defined as $\{x \mid x \in X \wedge \exists y \in Y\ (P(x, y))\}$

and the antijoin operation $X \triangleright_{x,y:P} Y$ as $\{x \mid x \in X \wedge \nexists y \in Y\,(P(x,y))\}$.)

Examples of predicates that may or may not be rewritten into the desired format are listed in Table 2. Predicates above the separation line are predicates that may occur in SQL (being a subset of TM); predicates below the separation line are specific TM predicates involving set-valued attributes.

$P(x,z)$	$P'(x,v)$
$z = \emptyset$	$\nexists v \in z\,(true)$
$z \neq \emptyset$	$\exists v \in z\,(true)$
$count(z) = 0$	$\nexists v \in z\,(true)$
$x.a = count(z)$	
$x.a \in z$	$\exists v \in z\,(v = x.a)$
$x.a \notin z$	$\nexists v \in z\,(v = x.a)$
$x.a \subset z$	
$x.a \supset z$	
$x.a \subseteq z$	
$x.a \supseteq z$	$\nexists v \in z\,(v \notin x.a)$
$x.a \not\supseteq z$	$\exists v \in z\,(v \notin x.a)$
$x.a = z$	
$x.a \cap z = \emptyset$	$\nexists v \in z\,(v \in x.a)$
$x.a \cap z \neq \emptyset$	$\exists v \in z\,(v \in x.a)$
$x.a \ni z$	
$\forall w \in x.a\,(w \subseteq z)$	
$\forall w \in x.a\,(w \supseteq z)$	$\nexists v \in z\,(\exists w \in x.a : (v \notin w))$

Table 2. Rewriting TM predicates

8 Query Processing Example

In this section, we illustrate our ideas by means of an example concerning an acyclic query with nesting in the WHERE clause in which correlation predicates are all neighbour predicates (having free variables declared in the immediately surrounding block).

In a preprocessing phase, predicates between query blocks are rewritten into calculus expressions if possible. The purpose of this rewrite phase is to determine whether nest join operations may be replaced by flat join operations (semi- or antijoin), as indicated in the previous section.

Nested queries are processed by performing a number of join operations and executing a number of function applications (for example projections) and selections on the join results. If predicates between query blocks require grouping, a nest join operator is applied; if predicates do not need grouping a flat join operation is executed. Replacement of a nest join operator by a semijoin or antijoin operator is advantageous because the semi- and antijoin can be implemented more efficiently than the nest (or regular) join

operator. Note that, like the semijoin, the antijoin operation may be implemented as a modification of the regular join. Consider the following query:

$$
\begin{array}{l}
\text{SELECT } x \\
\text{FROM } X\ x \\
\text{WHERE } x.a \subseteq \text{SELECT } y.a \quad (P_1) \\
\qquad\qquad \text{FROM } Y\ y \\
\qquad\qquad \text{WHERE } x.b = y.b \wedge \\
\qquad\qquad\qquad\qquad y.c \subseteq \text{SELECT } z.c \quad (P_2) \\
\qquad\qquad\qquad\qquad\qquad \text{FROM } Z\ z \\
\qquad\qquad\qquad\qquad\qquad \text{WHERE } y.d = z.d
\end{array}
$$

Predicates P_1 and P_2 between query blocks do require grouping (see Table 2), so we may have the following execution strategy:

(1) A nest join with operands Y and Z on join predicate $y.d = z.d$. Each element of Z satisfying the join predicate is projected on the c attribute. The result of this step is the set: $T_1 = \{y +\!+ \langle zs = \{z.c \mid z \in Z \wedge y.d = z.d\}\rangle \mid y \in Y\}$. Note zs is an arbitrary label.
(2) The result of (1) is restricted such that $y.c \subseteq y.zs$: $\{y \mid y \in T_1 \wedge y.c \subseteq y.zs\}$.
(3) The result of (2) is nest joined with X on join predicate $x.b = y.b$ and projected on attribute a. (A projection of (2) on attributes a and b may proceed the nest join operation.) We now have: $T_3 = \{x +\!+ \langle ys = \{y.a \mid y \in T_2 \wedge x.b = y.b\} \mid x \in X\}$. Again, label ys is arbitrary.
(4) Finally, the result of (3) is restricted such that $x.a \subseteq x.ys$ and projected on the attributes of X (attributes a and b): $T_4 = \{\langle a = x.a, b = x.b\rangle \mid x \in T_3 \wedge x.a \subseteq x.ys\}$.

Now assume that the operators $\subseteq$ in predicates P_1 and P_2 are changed in $\in$ and $\notin$, respectively, then the nest join operation in (1) may be replaced by an antijoin operation, and the nest join in (3) may be replaced by a semijoin operation. The additional join predicates are $y.c = z.c$ and $x.a = y.a$, respectively.

9 Conclusions and Future Work

As in relational systems supporting SQL, in complex object models supporting an SQL-like query language optimization of nested queries is an important issue. A naive way to handle nested queries is by nested-loop processing. However, it is better to transform nested queries into flat, or join queries, because join queries can be implemented in many different ways. In a complex object model, it is not always possible to flatten nested queries–in this paper, we have described a class of nested SFW-expressions that can be flattened without problem. For those nested queries that cannot be transformed into join queries we have introduced the nest join operator, allowing correct and efficient processing of general nested queries.

Future work concerns a number of issues. We need a formal algorithm to translate general SFW-query blocks of TM into the algebra, also taking into account nesting in the SELECT clause, multiple subqueries, and multiple nesting levels (including cyclic queries). Logical optimization (rewriting algebraic expressions) may follow the

translation process. Therefore, the algebraic properties of the nest join operator have to be further investigated, by analogy with for example the work of [11] concerning the outerjoin operator.

Acknowledgements

We thank our colleagues Rolf de By and Paul Grefen for comments on earlier drafts of this paper.

References

1. Bal, R. et al.: The TM Manual version 2.0. University of Twente, Enschede (1993)
2. Balsters, H., de By, R.A., and Zicari, R.: Typed Sets as a Basis for Object-Oriented Database Schemas. Proceedings ECOOP, Kaiserslautern (1993)
3. Balsters, H. and de Vreeze, C.C.: A Semantics of Object-Oriented Sets. Proceedings 3rd International Workshop on Database Programming Languages, Nafplion, Greece (1991)
4. Dayal, U.: Of Nests and Trees: A Unified Approach to Processing Queries That Contain Nested Subqueries, Aggregates, and Quantifiers. Proceedings VLDB, Brighton (1987)
5. Ganski, R.A. and Wong, A.K.T.: Optimization of Nested SQL Queries Revisited. Proceedings SIGMOD (1987)
6. Kiesling, W.: SQL-like and QUEL-like Correlation Queries with Aggregates Revisited. Memorandum UCB/ERL 84/75, Berkeley (1984)
7. Kim, W.: On Optimizing an SQL-like Nested Query. ACM Transactions on Database Systems **7** 3 (1982) 443–469
8. Muralikrishna, M.: Optimization and Dataflow Algorithms for Nested Tree Queries. Proceedings VLDB, Amsterdam (1989)
9. Muralikrishna, M.: Improved Unnesting Algorithms for Join Aggregate SQL Queries. Proceedings VLDB. Vancouver, 1992.
10. Pistor, P. and Andersen, F.: Designing a Generalized NF^2 Model with an SQL-type Language Interface. Proceedings VLDB, Kyoto (1986)
11. Rosenthal, A. and Galindo-Legaria, C.: Query Graphs, Implementing Trees, and Freely-Reorderable Outerjoins. Proceedings SIGMOD, Atlantic City (1990)
12. Schek, H.J. and Scholl, M.H.,: The Relational Model with Relation-Valued Attributes. Information Systems **11** 2 (1986) 137–147
13. Scholl, M.H.: Theoretical Foundations of Algebraic Optimization Utilizing Unnormalized Relations. Proceedings First International Conference on Database Theory. LNCS 243, Springer-Verlag (1986)