From Nested-Loop to Join Queries in OODB

Hennie J. Steenhagen Peter M.G. Apers Henk M. Blanken Rolf A. de By
Department of Computer Science, University of Twente
PO Box 217, 7500 AE Enschede, The Netherlands
{hennie,apers,blanken,deby}@cs.utwente.nl

Abstract

Most declarative SQL-like query languages for object-oriented database systems (OOSQL) are orthogonal languages allowing for arbitrary nesting of expressions in the **select-**, **from-**, and **where-**clause. Expressions in the **from-**clause may be base tables as well as set-valued attributes. In this paper, we propose a general strategy for the optimization of nested OOSQL queries. As in the relational model, the translation/optimization goal is to move from tuple- to set-oriented query processing. Therefore, OOSQL is translated into the algebraic language ADL, and by means of algebraic rewriting nested queries are transformed into join queries as far as possible. Three optimization options are described, and a strategy to assign priorities to options is proposed.

1 Introduction

To support technical applications like CAD/CAM, GIS etc, relational technology has its shortcomings. In these areas, the popularity of object-oriented technology is growing. First, from the field of programming languages, *persistent* programming languages like GemStone [Bretl et al. 89] came, later followed by object-oriented *database systems* such as O_2 [BaDK92] and HP OpenODB [Lyng91]. The historical background is still visible in the sense that too little attention has been paid to *ad hoc* query facilities and database design tools. A number of proposals for declarative query languages for extended NF^2 and object-oriented data models has appeared, e.g. [PiAn86, BaCD92, Catt93]. We will use the term OOSQL for the various proposals of SQL-like languages for OODB. In this paper, we concentrate on efficient support for ad hoc queries formulated in a declarative query language.

An OOSQL query facility is inherently more complex than one for SQL, because nesting is allowed in all clauses, i.e. in the **select-**, **from-**, and **where-**clause. Expressions in the **from-**clause, the query block operands, may be base tables as well as set-valued attributes. Also, the predicates that are used in the **where-**clause are more complex, because comparisons between set-valued attributes, or set-valued attributes and base table expressions are allowed. An example is the test whether two parts have an overlap in their sets of subparts.

Due to the complexity of OOSQL, the dominant strategy to handle nesting is to execute it by means of nested-loop processing, leaving no room for optimization. We distinguish between two types of nesting: nesting required to iterate over base tables and nesting required to iterate over set-valued attributes. Ideally, nesting over base tables should be translated to some kind of join so that a choice can be made between various efficient join implementations. Of course, this problem already occurred in SQL [Kim82]. In general, nested SQL queries can be translated to a join in the relational algebra to get a better execution than the nested-loop execution. However, not all SQL queries can be translated to a join due to loss of dangling tuples in the join, a phenomenon known as the COUNT-bug. In [StAB94] we have shown that the COUNT-bug is only a special case of a more general problem occurring in nested OOSQL queries.

The main focus of our research is to translate OOSQL queries to an extended relational algebra for complex objects, called ADL, to allow for an efficient

Proceedings of the 20th VLDB Conference
Santiago, Chile, 1994

execution. In this paper, we deal with the problem of trying to translate nested OOSQL queries to join queries in ADL, taking advantage of efficient implementations of join operators. We present a general approach for handling nesting in the **where**-clause, the **from**-, as well as in the **select**-clause; the discussion concentrates on nesting in the **where**-clause, though.

In the (extended) NF^2, as well as in the object-oriented literature, little work has been reported concerning efficient translation of SQL-like query languages into algebraic languages. For the NF^2 model, a translation of a calculus into an NF^2 algebra has been presented in [RoKS88], however, little attention has been paid to efficiency. Work has been done on the implementation of the extended NF^2 query language HDBL of the AIM project [SLPW89, SüLi90]; to our knowledge HDBL has not been translated into an algebra. In [ClMo93], a proposal for the optimization of nested O_2SQL queries has been made. O_2SQL is translated into an extension of the GOM algebra [KeMo93], and examples of optimization of nested algebra queries are given.

The organization of this paper is as follows. In Section 2 nesting in the various clauses of the select statement is introduced by means of examples, together with an example schema of an OODB. In Section 3 an algebra for complex objects called ADL is shown with a general translation of OOSQL queries to ADL queries. Section 4 addresses the problem of optimizing ADL queries. Three alternatives are discussed. Two of them, rewriting into relational join queries and the introduction of new operators are considered in Section 5 and Section 6. The paper ends in discussing future work.

2 Example

In this section, we give a simple example schema of the supplier-part database in OOSQL, together with some example queries.

```
Class Supplier with extension SUPPLIER
attributes
        sname : string,
        parts_supplied : { Part }
end Supplier

Class Part with extension PART
attributes
        pname : string,
        price : int,
        color : string
end Part

Class Delivery with extension DELIVERY
attributes
        supplier : Supplier,
        supply : { ⟨part : Part, quantity : int⟩ },
        date : date
end Delivery
```

The schema only shows the structural properties of the entities stored in the database; method and constraint definitions have been left out. Brackets ⟨ ⟩ and { } denote the tuple and set type constructor, respectively. Analogous to relational convention, we call the class extensions base tables.

Below we give example queries for nesting in the various clauses. With regard to the **where**-clause, one example is given for nesting over a base table, and another for nesting over a set-valued attribute.

Example Query 1 Nesting in the **select**-clause is used to produce set-valued attributes in a complex object.

Select the names of the suppliers together with the names of the red parts supplied:

```
select ⟨sname = s.sname,
        pnames = select p.pname
                 from p in s.parts_supplied
                 where p.color = "red"⟩
from s in SUPPLIER
```

Example Query 2 Nesting in the **from**-clause denotes query composition, that for example may occur as the result of expanding views or named intermediate tables.

Select all deliveries that concern supplier s_1 with date January 1, 1994:

```
select d
from d in (select e
           from e in DELIVERY
           where e.supplier.sname = "s1")
where d.date = 940101
```

Nesting in the **from**-clause is a type of nesting that does not pose problems with respect to translation/optimization; it can be removed easily.

Example Query 3 Nesting in the **where**-clause is used for restrictions.

1. Select the names of the suppliers supplying all parts supplied by supplier s_1:

```
select s.sname
from s in SUPPLIER
where s.parts_supplied ⊇ select t.parts_supplied
                         from t in SUPPLIER
                         where t.sname = "s1"
```

2. Select all deliveries that include red parts.

```
select d
from d in DELIVERY
where exists x in (select s
                   from s in d.supply
                   where s.part.color = "red")
```

In the first query, the operand of the inner **sfw**-block is the base table SUPPLIER: in the second the operand is the set-valued attribute supply. In the first, we have a set comparison between blocks, in the second a quantifier expression.

One important difference between SQL and OOSQL is that OOSQL is an *orthogonal* language. The expressions in the **from**- and **select**-clause of OOSQL may be arbitrary, also containing other **select-from-where** (**sfw**) expressions (subqueries), provided they are correctly typed. Predicates may also be built up from arbitrary expressions including quantifiers **forall** and **exists** and set comparison operators. The focus of this paper is a general strategy for dealing with nested OOSQL queries with nesting in the **select**- or **where**-clause. (Nesting in the **from**-clause is handled easily.) The discussion in subsequent sections will be centered around nested queries with nesting in the **where**-clause, however, techniques presented apply to nested queries with nesting in the **select**-clause as well.

Following the relational line of work, the goal in translation and optimization of OOSQL is to move from tuple- to set-oriented query processing. Our approach, as in [ClMo93], is to translate nested OOSQL queries into nested algebraic expressions, and then to try to rewrite nested algebraic expressions into join expressions. In the following section, we briefly present the algebraic language ADL.

3 The Complex Object Algebra ADL

The language ADL is a typed algebra for complex objects in the style of the NF^2 algebra of [ScSc86], allowing for nesting of expressions. Among the constructors supported are the tuple ($\langle\,\rangle$) and set ($\{\,\}$) type constructor; the basic type **oid** is used to represent object identity.

Roughly, the algebraic operators of the language ADL are the standard set (comparison) operators and multiple union (flatten), extended Cartesian product (in which operand tuples are concatenated) and division, the map operator α, selection σ, projection π, the renaming operator ρ, and the restructuring operators nest (ν) and unnest (μ). The map operator, well-known from functional languages and appearing under many different names in the literature, is used to apply a function to every element of a set. The function applied may be arbitrarily complex, so that the effect of a map operation may vary from a simple projection to the production of complex results. Furthermore, a number of join operators is supported: the regular join $\bowtie$, the semijoin $\ltimes$, and the antijoin $\triangleright$. The semijoin (a regular join followed by the projection on the left-hand join operand attributes) is a join operator that is useful in processing so-called tree queries [Kamb85]. The antijoin is defined as a semijoin followed by a set difference of the left-hand join operand and the semijoin result. The antijoin operator is less well-known than the semijoin operator; it can be employed to efficiently process tree queries involving universal quantification. In selections and joins arbitrarily complex predicates can be used, including predicates containing quantifiers. Of course aggregate functions are part of the language too. Below, we give the semantics of some of the ADL operators used in this paper. For presentation purposes, a simplified notation is used; it is assumed no attribute naming conflicts occur. The operator $\circ$ is used to denote tuple concatenation. The schema function SCH, when applied to a table expression, delivers the top level attribute names.

1. (Flatten) $\bigcup(e) = \{x \mid x \in X \wedge X \in e\}$
2. (Tuple subscription)
   $e[a_1, \ldots, a_n] = \langle a_1 = e.a_1, \ldots, a_n = e.a_n\rangle$
3. (Tuple "update")
   Let $\text{SCH}(e) = \{a_1, \ldots, a_n, b_1, \ldots, b_m\}$, then:
   e **except** $(a_1 = e_1, \ldots, a_n = e_n, c_1 = e'_1, \ldots, c_n = e'_k) = \langle a_1 = e_1, \ldots, a_n = e_n, b_1 = e.b_1, \ldots, b_m = e.b_m, c_1 = e'_1, \ldots, c_n = e'_k\rangle$
   The except operator may update existing tuple fields ($a_i = e_i$), leaving the remaining as they are ($b_i = e.b_i$), and may also extend the tuple with some new fields ($c_i = e'_i$).
4. (Map, or function application)
   $\alpha[x : f(x)](e) = \{f(x) \mid x \in e\}$
5. (Selection) $\sigma[x : p(x)](e) = \{x \in e \mid p(x)\}$
6. (Projection) $\pi_{a_1,\ldots,a_n}(e) = \{x[a_1, \ldots, a_n] \mid x \in e\}$
7. (Unnest) Let $\text{SCH}(e) = \{a, b_1, \ldots, b_m\}$, then:
   $\mu_a(e) = \{x' \circ x[b_1, \ldots, b_m] \mid x \in e \wedge x' \in x.a\}$
8. (Nest) Let $\text{SCH}(e) = \{a_1, \ldots, a_n, b_1, \ldots, b_m\}$, let $A = a_1, \ldots, a_n$, and let $B = b_1, \ldots, b_m$, then:
   $\nu_{A;a}(e) = \{x[B] \circ \langle a = X\rangle \mid x \in e \wedge X = \{x'[A] \mid x' \in e \wedge x'[B] = x[B]\}\}$
9. (Cartesian product)
   $e_1 \times e_2 = \{x_1 \circ x_2 \mid x_1 \in e_1 \wedge x_2 \in e_2\}$
10. (Regular join)
    $e_1 \underset{x_1,x_2:p(x_1,x_2)}{\bowtie} e_2 = \{x_1 \circ x_2 \mid x_1 \in e_1 \wedge x_2 \in e_2 \wedge p(x_1, x_2)\}$
11. (Semijoin)
    $e_1 \underset{x_1,x_2:p(x_1,x_2)}{\ltimes} e_2 = \{x_1 \mid x_1 \in e_1 \wedge \exists x_2 \in e_2 \bullet p(x_1, x_2)\}$
12. (Antijoin)
    $e_1 \underset{x_1,x_2:p(x_1,x_2)}{\triangleright} e_2 = \{x_1 \mid x_1 \in e_1 \wedge \nexists x_2 \in e_2 \bullet p(x_1, x_2)\}$

ADL operators that allow for nesting, the so-called *iterators*, are the map, select, and join operators, and also quantifiers. Iterators are operators having functions (lambda expressions $\lambda x.e$, denoted as $x : e$) as parameters; within the function body other operators may occur. Note that the parameter of join operators (the join predicate) is a function having two arguments; the function (denoted as $x_1, x_2 : p$) is written as a subscript to the join operator symbol. In the evaluation of joins, variables x_1 and x_2 are iterated over operands e_1 and e_2 respectively, and tuples are included in the result depending on the value of $p(x_1, x_2)$.

Mapping OOSQL to the algebra involves a mapping of types and a mapping of expressions. The mapping of *types* is carried out in the phase of logical database design. Here, we assume that each class extension is mapped to a table of (possibly complex) objects; a field of type **oid** is added to represent object identity, and class references are implemented by pointers, also of type **oid**. We also assume that class hierarchies are mapped to ADL types in some way or the other (the algebra does not support inheritance).

With respect to translation and optimization of *expressions*, we take the following approach. Translation of OOSQL queries into the algebra is done in a simple, almost one-to-one way. The algebra supports nesting of expressions, representing tuple-oriented, or nested-loop query processing, as well as a number of purely algebraic set-oriented operators. In the translation phase, nested OOSQL queries are translated into nested algebraic expressions. Following translation, in the phase of logical optimization, nested expressions are rewritten into set operations.

The OOSQL construct that does not have an immediate algebraic equivalent is the **sfw**-query block. In the translation phase, an **sfw**-query block is mapped to an algebraic expression consisting of a selection followed by a map:

$$\begin{aligned}&\textbf{select } e_1 \textbf{ from } x \textbf{ in } e_2 \textbf{ where } e_3\\ &\quad \equiv \alpha[x : e_1](\sigma[x : e_3](e_2))\end{aligned}$$

where σ computes the selection e_3 and α the "projection" e_1. In the select operation, variable x is iterated over operand e_2 and the operand is restricted according to the values of the **where**-clause predicate e_3; in the map operation α, variable x is iterated over the resulting operand subset, and for each of the tuples in this set the **select**-clause expression e_1 is evaluated.

To conclude this section, we formulate the goal in optimization of nested ADL queries. Operands of operators nested within parameter expressions of iterators may be either set-valued attributes or base table expressions. In this paper, the *goal* is to transform nested expressions, in which iterators having base tables as operands occur nested within parameter expressions of other iterators, into join expressions in which base tables occur *only at top level*. Of course the goal of unnesting applies to correlated subqueries[1] only; uncorrelated subqueries simply are constants, and treated as such. Assuming set-valued attributes are stored clustered, the unnesting of expressions with nested iterators having set-valued attributes as operands is not desirable. For example, $\sigma[x : \exists y \in Y \bullet p](X)$ with Y a base table, is transformed into the semijoin operation $X \ltimes_{x,y:p} Y$, but $\sigma[x : \exists y \in x.c \bullet p](X)$ with c a set-valued attribute stored with the X-tuples is left as it is. In short, the goal in translation/optimization is to remove base tables from the parameter expressions of iterators, moving from tuple- to set-oriented query processing.

4 Optimization Of Nested Algebra Queries

In this section we present a general approach to optimize nested ADL queries. Three optimization options are given, together with example queries. The section ends with a strategy to give a priority to the options.

The example queries given below concern the supplier-part database of which the OOSQL schema was given in Section 2. In ADL, the types of SUPPLIER and PART are as follows:

$$\begin{aligned}&\text{SUPPLIER} : \{\langle \mathit{sid} : \textbf{oid},\\ &\qquad \mathit{sname} : \textbf{string},\\ &\qquad \mathit{parts} : \{\langle \mathit{pid} : \textbf{oid}\rangle\}\rangle\}\\ &\text{PART} : \{\langle \mathit{pid} : \textbf{oid},\\ &\qquad \mathit{pname} : \textbf{string},\\ &\qquad \mathit{price} : \textbf{int},\\ &\qquad \mathit{color} : \textbf{string}\rangle\}\end{aligned}$$

We distinguish three ways of optimizing nested ADL queries: (1) the unnesting of attributes by using the unnest operator, (2) the unnesting of nested expressions by transforming them into relational join queries, and (3) using new operators that (analogous to for example the relational join) are defined especially to enhance performance. Below, we discuss the options in more detail.

Unnesting Of Attributes

If nesting is caused by iteration over a set-valued attribute it is possible to unnest this attribute. Depending on whether the result is nested or not, the nest operator has to be applied. The unnesting of attributes has some disadvantages. First, nest and unnest are each others inverse only for PNF relations (nested relations of which the atomic attributes recursively form a key) that have no empty set-valued

[1]Correlated subqueries are iterator expressions that use variables from iterators in which they are nested.

attributes [RoKS88]. Second, first unnesting and later nesting again will be expensive due to duplication of attribute values and overhead caused by restructuring. Therefore, we only use this option if the final nesting is not required, and empty set-valued attributes cause no problem. Consider the query:

Example Query 4 Select the identifiers of suppliers supplying non-existing parts (violating referential integrity).

$$\pi_{sid}(\sigma[s : \exists z \in s.parts \bullet \\ \nexists p \in \text{PART} \bullet z = p[pid]](\text{SUPPLIER}))$$

The set-valued attribute *parts* is not needed in the result, so the above query may be rewritten into the antijoin query:

$$\pi_{sid}(\mu_{parts}(\text{SUPPLIER}) \underset{s,p:s.pid=p.pid}{\triangleright} \text{PART})$$

Note that because z is existentially quantified, the loss of tuples with empty set-valued attribute *parts* causes no problem (existential quantification over the empty set delivers *false*).

Transformation into join queries

In some cases two or more consecutive levels of nesting can be replaced by a join, antijoin, or semijoin operator, reducing the number of levels of nesting. In the ideal case all nesting has disappeared. Query 5 below shows such an example.

Example Query 5 Select the suppliers supplying red parts.

$$\sigma[s : \exists z \in s.parts \bullet \exists p \in \text{PART} \bullet \\ z = p[pid] \land p.color = "red"](\text{SUPPLIER})$$

This query can be rewritten into the semijoin query:

$$\text{SUPPLIER} \ltimes_{s,p:p[pid] \in s.parts} \\ \sigma[p : p.color = "red"](\text{PART})$$

In Section 5 this option is discussed in detail.

Using Special Operators

The relational join is not really necessary for the expressive power of the relational algebra; it was introduced to allow for various efficient implementations. The same can of course be done in an algebra for complex objects. Quite often we encounter that an efficient execution of a query is prohibited if we stick to generally-accepted operators. Therefore, we expect that introducing new operators is really necessary to obtain an efficient implementation. In Section 6 some new operators are discussed. The following query cannot be rewritten into a relational join query:

Example Query 6 Select suppliers names together with the parts supplied.

$$\alpha[s : \langle sname = s.sname, parts\_suppl = \\ \sigma[p : p[pid] \in s.parts](\text{PART})\rangle](\text{SUPPLIER})$$

However, using the so-called nestjoin operator $\dashv$ (see Section 6), the nested query can be rewritten into an efficient set operation:

$$\pi_{sname,parts\_suppl} \\ (\text{SUPPLIER} \underset{s,p:p[pid] \in s.parts;parts\_suppl}{\dashv} \text{PART})$$

Note that each of the options above can be applied to the top level expression as well as to subexpressions thereof. Given these options for optimization of nested ADL queries, the rewrite strategy is as follows:

1. Try to rewrite to the various relational join operators (join, antijoin, or semijoin).

2. If the above is not possible, try to flatten set-valued attributes; if the nesting phase can be skipped, this may be a strategy worthwhile considering.

3. If the above is not possible, try to rewrite to one of the newly defined operators, because they were introduced to get a better performance compared to nested-loop processing.

4. If none of the above works, leave the query as it is, which means that it is executed by means of nested loops.

The options "rewriting into relational join queries" and "using special operators" are discussed in more detail in the following two sections.

5 Rewriting Into Flat Relational Algebra

In the previous section, we have discussed some options for processing nested ADL queries. In this section, we investigate one of them—the rewriting of nested expressions into join expressions without unnesting set-valued attributes. The discussion concentrates on rewriting nested select (σ) expressions, the algebraic equivalent of nested OOSQL queries with nesting in the **where**-clause. However, rewriting nested expressions into join expressions is a strategy that can be applied to nesting in the map operator as well; the section is concluded by briefly discussing an example of this type of nesting.

5.1 General Query Format

In this section, we discuss transformation of nested OOSQL queries with nesting in the **where**-clause *in the presence of set-valued attributes.* Nesting in the **where**-clause is an important (and only) type of nesting allowed in the flat relational model; in complex object models it is considered equally important. The goal here, as in [Kim82], is to rewrite nested queries into *join queries* without unnesting set-valued attributes, so that instead of performing a naive nested-loop execution, the optimizer may choose from a number of different join processing strategies.

The general format of a two-block OOSQL query with nesting in the **where**-clause is the following:

$$\begin{array}{l} \textbf{select } F(x) \\ \textbf{from } x \textbf{ in } X \\ \textbf{where } P(x, Y') \\ \qquad \textbf{with } Y' = \textbf{select } G(x,y) \\ \qquad\qquad\qquad \textbf{from } y \textbf{ in } Y \\ \qquad\qquad\qquad \textbf{where } Q(x,y) \end{array}$$

(The **with** construct, enabling local definitions, is used here for reasons of convenience.) In Section 3, we have given the equivalence rule for translating the **sfw**-block into the algebra:

$$\begin{array}{l} \textbf{select } e_1 \textbf{ from } x \textbf{ in } e_2 \textbf{ where } e_3 \\ \quad \equiv \alpha[x : e_1](\sigma[x : e_3](e_2)) \end{array}$$

so the algebraic equivalent of an **sfw**-expression is a selection σ followed by a general function application α to compute the "projection". The map operator α is needed to compute arbitrarily structured results, as opposed to standard projections in the flat relational model.

The general format of the two-block **sfw**-expression with nesting in the **where**-clause as shown above is:

$$\begin{array}{l} \alpha[x : F(x)](\sigma[x : P(x, Y')](X)) \\ \textbf{with } Y' = \alpha[y : G(x,y)](\sigma[y : Q(x,y)](Y)) \end{array}$$

For simplicity, the functions F and G are assumed to be identity, so we have the format:

$$\sigma[x : P(x, Y')](X) \textbf{ with } Y' = \sigma[y : Q(x,y)](Y)$$

The query above is a nested query involving nested iteration over a base table: the outer selection predicate contains a subquery, which is a selection on base table Y. We want to transform this nested query into a join query, i.e. a query having no subqueries with base table operands.

5.2 Set Comparison Operations

To illustrate our ideas, at first we concentrate on two-block nested expressions with set comparison operations between query blocks. We investigate two unnesting techniques: unnesting by rewriting into quantifier expressions, and unnesting by grouping, a technique well-known from the relational model [Kim82].

In Section 5.2.1, we show that, in some cases, queries having set comparison operators between query blocks can be transformed into join queries by rewriting the set comparison operator into a quantifier expression. However, in other cases, rewriting into quantifiers has a negative effect on performance; other solutions have to be sought. The results of Section 5.2.1 are generalized to a rewrite heuristic for transforming two-block select queries with arbitrary quantifier expressions between blocks.

To the queries that cannot be unnested by rewriting into quantifier expressions, we apply the methods of [Kim82, GaWo87] for unnesting relational queries with aggregate functions between blocks. We show that the methods of [Kim82, GaWo87] are general techniques for transforming nested queries into join queries, however, to be of good use in complex models, they have to be adapted. We will do so by defining a new algebraic operator, the *nestjoin operator*, in Section 6.

We now continue by describing the query format of interest in this section. Assume that predicate P involves some set comparison operation relating attribute c of the outer operand X and the set Y', the subquery result. More formally, the query format is:

$$\sigma[x : x.c\ \theta\ Y'](X) \textbf{ with } Y' = \sigma[y : Q(x,y)](Y)$$

with $\theta \in \{\in, \subset, \subseteq, =, \supseteq, \supset, \ni\}$. Note that the type of attribute c varies with the comparison operator used; the type may be atomic ($\in$), or an arbitrary set type. If the operator used is $\ni$, c has a set-of-set type.

5.2.1 Unnesting By Rewriting Into Quantifier Expressions

In this section we show that translating set comparison operators into quantifier expressions offers possibilities to unnest nested queries. In [CeGo85], presenting a translation from SQL to the relational algebra, set comparison operators are dealt with by rewriting them into quantifier expressions in a preprocessing phase. Nested relational queries with quantifiers are easily translated into relational algebra operations. Existential quantification is mapped to a projection on a join (or product); universal quantification is handled by means of the division operator [Codd72].

For example, from the relational model we know that a set membership predicate can be translated into an existential subquery that is easily translated into a (semi)join operation. Let $q \equiv Q(x,y)$, then:

Table 1: Rewriting Set Comparison Operations

set comparison operation		quantifier expression
$x.c \in Y'$	$\equiv$	$\exists y \in Y' \bullet y = x.c$
$x.c \subset Y'$	$\equiv$	$x.c \subseteq Y' \land x.c \not\supseteq Y'$
	$\equiv$	$(\forall z \in x.c \bullet \exists y \in Y' \bullet z = y) \land (\not\forall y \in Y' \bullet y \in x.c)$
$x.c \subseteq Y'$	$\equiv$	$\forall z \in x.c \bullet \exists y \in Y' \bullet z = y$
$x.c = Y'$	$\equiv$	$x.c \subseteq Y' \land x.c \supseteq Y'$
	$\equiv$	$(\forall z \in x.c \bullet \exists y \in Y' \bullet z = y) \land (\forall y \in Y' \bullet y \in x.c)$
$x.c \supseteq Y'$	$\equiv$	$\forall y \in Y' \bullet y \in x.c$
$x.c \supset Y'$	$\equiv$	$x.c \supseteq Y' \land x.c \not\subseteq Y'$
	$\equiv$	$(\forall y \in Y' \bullet y \in x.c) \land (\not\forall z \in x.c \bullet \exists y \in Y \bullet z = y)$
$x.c \ni Y'$	$\equiv$	$\exists z \in x.c \bullet z = Y'$

Rewriting Example 1 SET MEMBERSHIP

$$\begin{aligned}
&\sigma[x : x.c \in \sigma[y : q](Y)](X)\\
&\equiv \sigma[x : \exists y \in \sigma[y : q](Y) \bullet y = x.c](X)\\
&\equiv \sigma[x : \exists y \in Y \bullet y = x.c \land q](X)\\
&\equiv X \ltimes_{x,y:y=x.c\land q} Y
\end{aligned}$$

First the operator $\in$ is rewritten into an existential quantification. Next, the select operation is removed from the operand (the range expression) of the existential quantifier, providing the possibility to translate the existential subquery into a semijoin operation in the last rewrite step. In this last step the actual unnesting is performed; the preceding rewrite steps are necessary to transform the input expression into the format suitable for unnesting. The equivalence rules for unnesting quantifier expressions nested within select operators are the following.

Rule 1 UNNESTING QUANTIFIER EXPRESSIONS Let X and Y be table expressions, and let x not be free in Y, then:

1. $\sigma[x : \exists y \in Y \bullet p](X) \equiv X \ltimes_{x,y:p} Y$
2. $\sigma[x : \nexists y \in Y \bullet p](X) \equiv X \triangleright_{x,y:p} Y$

A nested query with existential quantification is translated into a semijoin operation; negated existential (i.e. universal) quantification is dealt with by means of the antijoin operator.

The same method, rewriting set comparison operators by means of quantifiers, can be applied in complex object models as well. Again, let $q \equiv Q(x, y)$, and consider the following example dealing with the set inclusion operator:

Rewriting Example 2 SET INCLUSION

$$\begin{aligned}
&\sigma[x : \sigma[y : q](Y) \subseteq x.c](X)\\
&\equiv \sigma[x : \forall y \in \sigma[y : q](Y) \bullet y \in x.c](Y)](X)\\
&\equiv \sigma[x : \nexists y \in \sigma[y : q](Y) \bullet y \notin x.c](X)\\
&\equiv \sigma[x : \nexists y \in Y \bullet q \land y \notin x.c](X)\\
&\equiv X \underset{x,y:q\land y\notin x.c}{\triangleright} Y
\end{aligned}$$

Table 2: Rewriting Predicates

$P(x,Y')$	quantifier expression
$Y' = \emptyset$	$\nexists y \in Y' \bullet true$
$count(Y') = 0$	$\nexists y \in Y' \bullet true$
$x.c \cap Y' = \emptyset$	$\nexists y \in Y' \bullet y \in x.c$
$\forall z \in x.c \bullet z \supseteq Y'$	$\nexists y \in Y' \bullet \exists z \in x.c \bullet y \notin z$

In the example above, the same procedure as in Rewriting Example 1 is followed (rewriting into quantification, transformation of the range expression, and unnesting). In addition, the universal quantifier is transformed into a negated existential quantifier by pushing through negation to enable transformation into the antijoin operation.

All set comparison operators can be rewritten into quantifier expressions, as shown in Table 1. In the table, in all cases except for the last we have expanded operators until quantification(s) over Y' take(s) place. We see that expanding operators $\in$ and $\supseteq$ leads to a (negated) existential quantifier expression that is suited for unnesting by applying Rule 1; expansion of the other operators leads to a multiple subquery expression, that cannot be unnested that way. Note that negation of the set comparison operation does not influence the possibilities for unnesting. Negating the operator negates the quantifier expression; antijoins are used instead of semijoins and vice versa.

So far, we restricted our discussion to two-block nested queries with predicates of the form $x.c\ \theta\ Y'$, with θ a set comparison operator. In Table 2, we show some more examples of predicates that can be rewritten into (negated) existential quantification, the form suitable for transformation in relational join expressions. To determine exactly which types of predicates can be rewritten is a topic of future research.

Rewrite Heuristic

From the discussion above, an important rewrite heuristic for nested expressions with predicates consisting of arbitrary quantifier expressions can be derived. Difficulties with unnesting arise whenever subqueries with base tables as operands are nested within iterators with set-valued attributes as operands, and the order of nesting cannot be changed. Consider the last expression of Table 2. We have:

Rewriting Example 3 EXCHANGING QUANTIFIERS

$$\begin{aligned} & \forall z \in x.c \bullet z \supseteq Y' \\ \equiv\ & \forall z \in x.c \bullet \forall y \in Y' \bullet y \in z \\ \equiv\ & \forall y \in Y' \bullet \forall z \in x.c \bullet y \in z \\ \equiv\ & \nexists y \in Y' \bullet \exists z \in x.c \bullet y \notin z \end{aligned}$$

By expanding the comparison operator and exchanging universal quantifiers, the predicate is put in a form suitable for unnesting according to Rule 1. The general rewrite heuristic is formulated as follows. Let P be a quantifier expression in Prenex Normal Form:

$$P \equiv \forall/\exists x_1 \in e_1 \bullet \forall/\exists x_2 \in e_2 \ldots \forall/\exists x_n \in e_n \bullet p$$

in which the range expressions e_i are either base tables or set-valued attributes. To enable unnesting of (sub)expressions, the goal is to move quantification over base tables to the left of the quantifier expression. This goal may be achieved by exchanging universal or existential quantifiers.

5.2.2 Unnesting By Grouping

Another way to deal with set comparison operators is to use grouping. In [Kim82, GaWo87], grouping is used in transforming nested queries with aggregate functions between query blocks. As we will see, the method of [GaWo87] in fact represents a general way of treating nested queries that can be applied in complex object models as well[2]. However, in some cases the results achieved are not correct due to the loss of dangling tuples in the relational join operation.

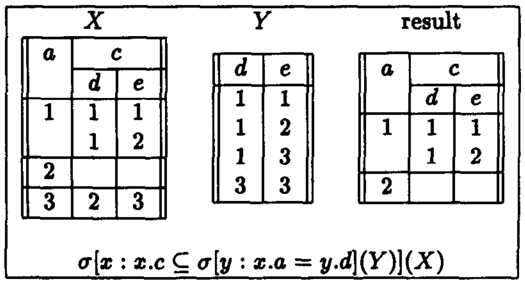
In [Kim82, GaWo87], methods for unnesting queries with aggregate functions between query blocks of the form $\sigma[x : P(x, agg(\sigma[y : Q(x,y)](Y)))](X)$ are presented. Below, we apply the method of [GaWo87] to nested queries with set comparators between blocks.

Consider the following nested query, an example of which is given in Figure 1.

$$\sigma[x : x.c \subseteq \sigma[y : x.a = y.d](Y)](X)$$

Applying the transformation technique of [GaWo87] to the expression above, we have, in our own formalism:

[2]The method of [Kim82] can be applied only when the correlation predicate (or join predicate) is equality.



X

a	c	
	d	e
1	1	1
	1	2
2		
3	2	3

Y

d	e
1	1
1	2
1	3
3	3

result

a	c	
	d	e
1	1	1
	1	2
2		

$\sigma[x : x.c \subseteq \sigma[y : x.a = y.d](Y)](X)$

Figure 1: Nesting Involving Set-Valued Attribute

$$\pi_X(\sigma[x : x.c \subseteq x.ys](\nu_{Y;ys}(X \underset{x,y:x.a=y.d}{\bowtie} Y)))$$

The nested query is transformed into a flat join query consisting of (1) a join to evaluate the inner query block predicate, (2) a nest operation for grouping, (3) a selection for evaluating P, the predicate between blocks, and (4) a final projection. Example tables X and Y and the intermediate results of the join, nest, and project/select operation are shown in Figure 2.

We note that, as with relational queries involving the COUNT function, in the join query some kind of bug occurs, due to the loss of dangling tuples in the join; in analogy with the phrase "COUNT bug", we call this bug the "Complex Object bug". In the example, the tuple $\langle a = 2, c = \emptyset \rangle$ in X is not matched by any of the tuples $y \in Y$, so the subquery result is empty. In the join, this tuple is lost; in the nested query, the expression $\emptyset \subseteq \emptyset$ evaluates to *true*, so the tuple has to be included.

Now consider a variant of the query above, in which $\subseteq$ is changed into $\supseteq$:

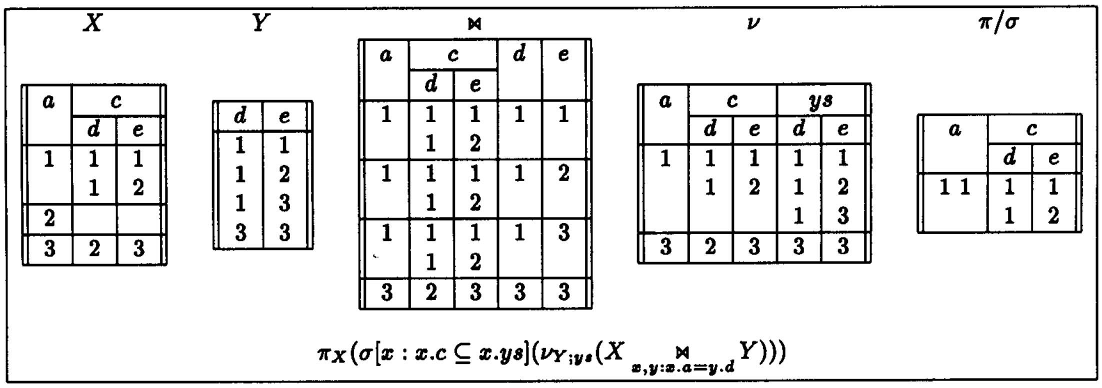
$$\sigma[x : x.c \supseteq \sigma[y : x.a = y.d](Y)](X)$$

Here as well, applying the same unnesting technique yields a Complex Object bug. All tuples $x \in X$ for which it holds that the subquery Y' is equal to the empty set should be included into the result, but are lost in the join.

In Table 3, we have listed the set comparison operations under consideration, together with the value of the predicate for subqueries delivering empty sets (a question mark meaning run-time dependence). Negated predicates are treated in the same way.

We have the following result.

- The relational transformation technique of [GaWo87] for unnesting nested queries having aggregate functions between query blocks, using grouping, may be applied to nested complex object queries involving set comparison operators between query blocks as well. However, in some cases the loss of dangling outer operand tuples in the join causes incorrect results.



X

a	c	
	d	e
1	1	1
	1	2
2		
3	2	3

Y

d	e
1	1
1	2
1	3
3	3

$\bowtie$

a	c		d	e
	d	e		
1	1	1	1	1
	1	2		
1	1	1	1	2
	1	2		
1	1	1	1	3
	1	2		
3	2	3	3	3

ν

a	c		ys	
	d	e	d	e
1	1	1	1	1
	1	2	1	2
			1	3
3	2	3	3	3

π/σ

a	c	
	d	e
1 1	1	1
	1	2

$$\pi_X(\sigma[x : x.c \subseteq x.ys](\nu_{Y;ys}(X \underset{x,y:x.a=y.d}{\bowtie} Y)))$$

Figure 2: The Complex Object Bug

Table 3: Set Comparison Operators And Bugs

$P(x,Y')$	$P(x,\emptyset)$
$x.c \in Y'$	*false*
$x.c \subset Y'$	*false*
$x.c \subseteq Y'$	?
$x.c = Y'$	?
$x.c \supseteq Y'$	*true*
$x.c \supset Y'$	?
$x.c \ni Y'$	?

- The value of the expression $P(x, Y')$, with the empty set substituted for Y', determines whether or not dangling tuples should be included into the result. Whenever $P(x, \emptyset)$ can be reduced statically to *true*/*false*, all/none of the dangling tuples $x \in X$ must be included into the result; whenever this value is undetermined at compile time, it is run-time dependent whether or not dangling tuples $x \in X$ should be included (cf. the predicate $x.c = count(Y')$). In other words, the unnesting technique used here is guaranteed to deliver correct results only if $P(x, \emptyset)$ can be statically reduced to *false*.

If not for the occurrence of bugs, the techniques of [Kim82, GaWo87] can be applied to nested queries with arbitrary predicates between blocks. One way to solve the COUNT bug in the relational model is to employ the outerjoin operator [GaWo87]. In using the outerjoin, NULL values are used to represent the empty set. This method may be applied in a slightly adapted way in complex object models as well. Another way to solve the COUNT bug is to use a binary aggregation operator [OOMa87, Naka90]. In Section 6, we discuss a new operator for unnesting nested queries that is based on the idea underlying binary aggregation, but separates predicate evaluation from join and grouping.

5.3 Nesting In The Map Operator

To conclude, we give another example of the strategy of rewriting nested expressions into relational join expressions, but now concerning nesting in the map operator (i.e. in the **select**-clause). The following equivalence rule can be used to transform a nested map operation into a join query:

Rule 2 NESTING IN THE MAP OPERATOR

$$\bigcup(\alpha[x : \alpha[y : x \circ y](\sigma[y : p])(Y)](X) \equiv X \bowtie_{x,y:p} Y$$

The nested map operation on the left hand side creates a set of sets that is flattened immediately afterwards; the same result is achieved by the right hand join expression.

Briefly summarizing this section, we have seen that (1) rewriting predicates into quantifier expressions may enable the transformation of nested expressions involving set-valued attributes into relational join expressions, (2) unnesting by grouping is a transformation technique that is generally applicable, if not for the occurrence of bugs. In the next section, we show how to avoid the occurrence of bugs by using the nestjoin operator; the general transformation strategy then is to transform nested queries into nestjoin expressions, but to use relational join operators whenever possible.

6 New Algebraic Operators

In this section, we give three examples of new algebraic operators that are well-suited for efficient implementation of nested OOSQL queries. Generally speaking, it is worthwhile to define new logical algebra operators whenever there can be found new access algorithms (or physical algebra operators [Grae93]) that are an improvement over nested-loop query processing. For example, the join can be implemented as an index nested-loop join, a sort-merge join, a hash join, etc.

In this section, we give some examples of operators that might be of use for improving performance in OO query processing. The first operator to be discussed is the nestjoin operator, defined in [StAB94] for the processing of nested queries requiring grouping. The second operation to be discussed is the PNHL algorithm of [DeLa92], useful for materializing set-valued attributes, and the third is the materialize operator of [BlMG93].

6.1 The Nestjoin Operator—Grouping During Join

In the previous section we have shown that unnesting by using grouping is a transformation strategy generally applicable, if not for the occurrence of bugs due to the loss of dangling tuples in the join. In [StAB94], we have defined an operator that combines grouping and join without losing dangling left operand tuples: the *nestjoin* operator. The nestjoin operator is to be used for the unnesting of nested queries that cannot be rewritten into flat relational join operations.

The nestjoin operator, denoted by the symbol $\dashv$, is a simple modification of the join operator. Instead of producing the concatenation of every pair of matching tuples, each left operand tuple is concatenated with the *set* of matching right operand tuples. To implement the nestjoin, common join implementation methods like the sort-merge join, or the hash join can be adapted. The definition of the nestjoin is as follows.

Definition 1 THE NESTJOIN OPERATOR (SIMPLE)

$$e_1 \underset{x_1,x_2:p(x_1,x_2);a}{\dashv} e_2 = \{x_1 \circ \langle a = X\rangle \mid x_1 \in e_1 \land X = \{x_2 \mid x_2 \in e_2 \land p(x_1,x_2)\}\} \quad (a \notin \text{SCH}(e_1))$$

Variables x_1 and x_2 are iterated over operands e_1 and e_2, respectively. Each left operand tuple $x_1 \in e_1$ is concatenated with the unary tuple $\langle a = X\rangle$, in which the set X contains those right hand operand tuples $x_2 \in e_2$ for which the predicate $p(x_1, x_2)$ holds. An example of the nestjoin operation is given in Figure 3, where relations X and Y are equijoined on the second attribute.

The nestjoin operator as defined above can be used for the transformation of two-block select expressions with arbitrary predicates between blocks. The simplified version of the two-block select query:

$$\sigma[x : P(x, Y')](X)) \textbf{ with } Y' = \sigma[y : Q(x,y)](Y)$$

is transformed into the nestjoin expression:

$$\pi_X(\sigma[z : P'](X \dashv_{x,y:Q(x,y);ys} Y))$$

In the nestjoin operation, for each tuple $x \in X$ the set of tuples $y \in Y$ is restricted according to predicate Q.

X

a	b
1	1
1	2
2	3

Y

c	d
1	1
2	1
3	3

result

a	b	ys: c	ys: d
1	1	1	1
		2	1
1	2		
2	3	3	3

$X \dashv_{x,y:x.b=y.d;ys} Y$

Figure 3: Nestjoin Example

In the selection, the nestjoin result is restricted according to predicate P'. Predicate P has to be adapted by substituting $z[X]$ (nestjoin tuple z projected on its X attributes) and $z.ys$ (the subquery result as attribute of nestjoin tuples z) for x and Y', respectively, i.e. $P' \equiv P(x, Y')[z[X]/x, z.ys/Y']$. A projection on the attribute values of X completes the computation.

The nestjoin operation can be used to process queries with nesting in the **select**- or **where**-clause. Queries having subqueries in the **select**-clause often denote nested results, so processing by means of the nest join operation will be appropriate. The general format of a query with nesting in the **select**-clause is:

```
select $F(x, Y')$
    with $Y'$ = select $G(x, y)$
                from $y$ in $Y$
                where $Q(x, y)$
from $x$ in $X$
where $P(x)$
```

Assume function G and predicate P are identity, then in the algebra we have:

$$\alpha[x : F(x, Y')](X)) \textbf{ with } Y' = \sigma[y : Q(x,y)](Y)$$

which is equivalent to:

$$\alpha[z : F'](X \dashv_{x,y:Q(x,y);ys} Y)$$

in which function F is adapted by performing the necessary substitutions:

$$F' \equiv F(x, Y')[z[X]/x, z.ys/Y']$$

Above, we have given a simplified definition of the nestjoin operator. For the transformation of general nested queries with deeper nesting levels, the nestjoin needs as an extra parameter a function to be applied to the right hand operand tuples [StAB94].

6.2 Materializing Set-Valued Attributes

Below, we describe two proposals for the materialization of (set-valued) attributes, in complex object models an operation presumed to occur frequently. In the first proposal of [DeLa92], an algorithm was given without defining a corresponding logical algebra operator; in [BlMG93], both a logical and a corresponding physical algebra operation are described.

The PNHL Algorithm

Below, we describe the algorithm of [DeLa92] for efficiently processing a nested expression in which a set-valued attribute is joined with a base table. This algorithm can be considered as a new physical algebra operation. Though the correspondence between logical and physical algebra operators usually is not one-to-one [Grae93], the question is whether it is useful to define new logical operators for algorithms such as that of [DeLa92]. The following query expresses a nested natural join ($\star$) operation:

$$\alpha[x : x \textbf{ except } (parts = x.parts \star_{z,y:z.pid=y.pid} \text{PART})](\text{SUPPLIER})$$

In [DeLa92], a hash-based algorithm called Partitioned Nested-Hashed-Loops (PNHL) algorithm for computing this type of join operation is described and performance measures are reported. The algorithm builds a hash table for those segments of operand PART that fit into main memory and then probes operand SUPPLIER against each segment of the hash table, thus building partial results. Partial results are merged in the second phase of the algorithm. Compared to the unnest-join-nest processing method, the algorithm achieves better performance. Comparing the PNHL algorithm with traditional hash join, we see that in the PNHL algorithm, only the flat table can be the build table (the inner operand PART in the example), whereas in relational hash join usually the smaller operand is chosen as build table.

The Materialize Operator

In object-oriented database systems the concepts of object identity and path expressions play an important role. Object identifiers can be implemented either as physical or as logical pointers. Implementing object-identifiers as physical pointers opens the way to new join implementation methods (pointer-based joins, [ShCa90]).

Also, object identifiers can be usefully employed to implement path expressions, i.e. the user-defined relationships or links between object classes. In [BlMG93], path expressions are represented by the operator materialize. Materialize is defined as a new logical algebra operator, with the purpose to explicitly indicate the use of inter-object references, i.e. to indicate where path expressions are used and where therefore algebraic transformations can be applied. The operator is implemented by an access algorithm called assembly, a generalization of the concept of a pointer-based join.

7 Conclusion And Future Work

As in relational systems supporting SQL, in OO data models supporting an SQL-like query language (OOSQL), optimization of nested queries is an important issue. A naive way to handle nested queries is by nested-loop processing (tuple-oriented query processing), however, it is better to transform nested queries into join queries, because join queries can be implemented in many different ways (set-oriented query processing).

In this paper, we have presented a general approach to optimization of nested OOSQL queries. In OOSQL, nesting may occur in the **where**-, **from**-, and **select**-clause. An additional complication in complex object models is the support for iteration over set-valued attributes. The goal is to transform nested OOSQL queries having correlated subqueries with base table expressions as operands into join queries in which base tables occur only at top level. First, we try to rewrite nested expressions into relational join operations. Second, we consider whether the unnesting of set-valued attributes is a possible (for theoretical reasons) and a worthwhile (for reasons of performance) optimization option. Third, if the previous steps do not give the result wanted, we use new operators especially defined to improve performance. Finally, if none of the previous steps work, we resort to nested-loop processing.

We have shown that transformation of nested OOSQL queries dealing with set-valued attributes into relational join queries is not always possible. In many cases, the unnesting of nested OOSQL queries requires some form of grouping in the unnested, or join query. Relational transformation techniques for nested queries requiring grouping (nested queries with aggregate functions between blocks) do not always give correct results; to improve matters we have defined a new operator called the nestjoin operator.

Future work concerns a number of issues. First, we need a precise characterization of nested queries requiring grouping or not. Second, for those queries that do require grouping, new implementation techniques have to be investigated. Third, new features characteristic of OO data models, like object identity and path expressions, provide new opportunities to improve performance. At the logical as well as the physical algebra level new operators may be defined and implemented. Finally, the ultimate goal of course is a general (syntax-driven) translation/optimization algorithm for arbitrary nested OOSQL queries, including queries with multiple subqueries and multiple nesting levels.

References

[BaCD92] Bancilhon, F., S. Cluet, and C. Delobel, *A Query Language for O_2*, in *Building an Object-Oriented Database System—The Story of O_2*, eds. F. Bancilhon, C. Delobel, and P. Kannelakis, Morgan Kaufmann Publishers, San Mateo, California, 1992.

[BaDK92] Bancilhon, F., C. Delobel, and P. Kannelakis (eds.), *Building an Object-Oriented Database System—The Story of O_2*, Morgan Kaufmann Publishers, San Mateo, California, 1992.

[BlMG93] Blakeley, J.A., W.J. McKenna, and G. Graefe, "Experiences Building the Open OODB Optimizer" *Proceedings ACM SIGMOD*, 1993.

[Bretl et al. 89] Bretl, R. et al., "The GemStone Database Management System," in *Object-Oriented Concepts, Databases, and Applications*, eds. W. Kim and F.H. Lochovsky, Addison-Wesley, Reading, MA, 1989.

[Catt93] R.G.G. Cattell, ed., *The Object Database Standard: ODMG-93*, Morgan Kaufman Publishers, San Mateo, California, 1993.

[CeGo85] Ceri, S. and G. Gottlob, "Translating SQL into Relational Algebra: Optimization, Semantics, and Equivalence of SQL Queries," *IEEE Transactions on Software Engineering*, 11(4), April 1985.

[ClMo93] Cluet, S. and G. Moerkotte, "Nested Queries in Object Bases," *Proceedings Fourth International Workshop on Database Programming languages*, New York, Sept. 1993.

[Codd72] Codd, E.F., "Relational Completeness of Data Base Sublanguages," In *Data Base Systems*, ed. R. Rustin, Prentice Hall, 1972.

[DeLa92] Desphande, V. and P.-A. Larson, "The Design and Implementation of a Parallel Join Algorithm for Nested Relations on Shared-Memory Multiprocessors," *Proceedings IEEE Conference on Data Engineering*, pp. 68-77, Tempe, Arizona, February 1992.

[GaWo87] Ganski, R.A. and A.K.T. Wong, "Optimization of Nested SQL Queries Revisited," *Proceedings ACM SIGMOD*, 1987.

[Grae93] Graefe, G., "Query Evaluation Techniques for Large Databases," *ACM Computing Surveys*, 25(2), pp. 73-170, June 1993.

[Kamb85] Kambyashi, Y., "Cyclic Query Processing," in *Query Processing in Database Systems*, eds. W. Kim, D.S. Reiner, and D.S. Batory, Springer Verlag, pp. 62-78, 1985.

[KeMo93] Kemper, A. and G. Moerkotte, "Query Optimization in Object Bases : Exploiting Relational Techniques," in *Query Processing for Advanced Database Systems*, eds. J.-C. Freytag, D. Maier, and G. Vossen, Morgan Kaufman Publishers, San Mateo, California, 1993.

[Kim82] Kim, W., "On Optimizing an SQL-like Nested Query," *ACM TODS*, 7(3), pp. 443-469, September 1982.

[Lyng91] Lyngbaek, P., "From Relational Databases to Objects and Beyond," in *Advances in Data Management*, eds. P. Sadanandan, T.M. Vijayaraman, McGraw-Hill Publishing Company Ltd, 1991.

[Naka90] Nakano, R. " Translation with Optimization from Relational Calculus to Relational Algebra Having Aggregate Functions," *ACM TODS*, 15(4), pp. 518-557, December 1990.

[OOMa87] Ozsoyoglu, G., Z.M. Ozsoyoglu, and V. Matos, "Extending Relational Algebra and Relational Calculus with Set-Valued Attributes and Aggregate Functions," *ACM TODS*, 12(4), pp. 566-592, December 1987.

[PiAn86] Pistor, P. and F. Andersen, "Designing a Generalized NF^2 Model with an SQL-Type Language Interface," *Proceedings VLDB*, Kyoto, August 1986.

[RoKS88] Roth, M.A., H.F. Korth, and A. Silberschatz, "Extended Algebra and Calculus for Nested Relational Databases," *ACM TODS*, 13(4), pp. 389-417, December 1988.

[SLPW89] Saake, G., V. Linneman, P. Pistor, and L. Wegner, "Sorting, Grouping and Duplicate Elimination in the Advanced Information Management Prototype," *Proceedings VLDB*, Amsterdam, 1989.

[ScSc86] Schek, H.-J. and M.H. Scholl, "The Relational Model with Relation-Valued Attributes," *Information Systems*, 11(2), pp. 137-147, 1986.

[ShCa90] Shekita, E.J. and M.J. Carey, "A Performance Evaluation of Pointer-Based Joins," *Proceedings ACM SIGMOD*, pp. 300-311, Atlantic City, May 1990.

[StAB94] Steenhagen, H.J., P.G.M. Apers, and H.M. Blanken, "Optimization of Nested Queries in a Complex Object Model," *Proceedings EDBT*, Cambridge, March 1994.

[SüLi90] Südkamp, N. and V. Linnemann, "Elimination of Views and Redundant Variables in an SQL-like Database Language for extended NF^2 Structures," *Proceedings VLDB*, Brisbane, 1990.