Translating OSQL Queries into Efficient Set Expressions

Hennie J. Steenhagen Rolf A. de By Henk M. Blanken

Department of Computer Science, University of Twente
PO Box 217, 7500 AE Enschede, The Netherlands
{hennie,deby,blanken}@cs.utwente.nl

Abstract. Efficient query processing is one of the key promises of database technology. With the evolution of supported data models—from relational via nested relational to object-oriented—the need for such efficiency has not diminished, and the general problem has increased in complexity.
In this paper, we present a heuristics-based, extensible algorithm for the translation of object-oriented query expressions in a variant of OSQL to an algebra extended with specialized join operators, designed for the task. We claim that the resulting algebraic expressions are cost-efficient.
Our approach builds on well-known optimization strategies for the relational model, but extends them to include relations and more arbitrary sets as values. We pay special attention to the most costly forms of OSQL queries, namely those with full subqueries in the SELECT- or WHERE-clause. The paper builds on earlier results [17, 18].

1 Introduction

Currently, the ODMG group is working on the standards for object-oriented database management systems [3]; the ODMG proposal includes a description of an object query language named OQL, which is an SQL-like language. How to implement such a language, i.e. the subject of efficient query processing in object-oriented database systems, is an important research topic. This paper studies the translation of a prototype object-oriented SQL language (OSQL) into an algebra supporting complex objects. We believe that, as for the relational model, set-orientation is an appropriate query processing paradigm for object-oriented models also. Set operators allow to apply techniques such as sorting or hashing to improve performance. The goal is to obtain algebraic expressions that have good performance.

Important features of object-oriented data models are object identity, inheritance, the presence of complex objects, and the possibility to define methods. In our opinion, SQL languages for object-oriented models can be considered as an extension of SQL languages for extended nested relational models [15]. Common features are the presence of complex objects and the orthogonality of language design. In the implementation of OSQL, precisely these features are of major importance. The work presented here is meant to serve as the basis for the implementation of OSQL. Specific object-oriented features can be handled as an addition or an extension. For example, the presence of object identity allows to speed up join algorithms [16]. We remark that, in our framework [1], methods are written using OSQL instead of some general purpose programming language. Hence, method calls in a query can be textually substituted by their OSQL definition, allowing for additional optimization.

We study the transformation of nested OSQL queries. In [17], we showed that in complex object models it is impossible to transform arbitrary nested queries into flat join queries. As a solution, we introduced the nestjoin operator. In [18], we described a general approach to handle nested queries; here, we are more concrete and present a translation algorithm. Related to our work is that of [5], in which optimization of nested O_2SQL queries is discussed.

In Section 2, we briefly describe the language used and in Section 3 we discuss our approach to the transformation of nested OSQL queries. Next, in Section 4, the main steps of the algorithm are described, and a basic set of rewrite rules and an initial rewrite strategy is given. It becomes clear that, in order to obtain an efficient result, (multi-variable) parameter expressions have to be split. Heuristics are needed to guide the process of splitting expressions; these are presented in Section 5. In Section 6, we compare our work with that of others and, finally, Section 7 gives conclusions and discusses future work.

2 Preliminaries

We work within one language. The type system of our language is that of the nested relational model, extended in the sense that, besides relation-valued attributes, arbitrary set-valued attributes are allowed as well. The language consists of SQL-like constructs such as collect and quantifiers, which allow for nesting (the OSQL part of the language), and of pure algebraic operators such as set operators and join (the logical algebra part of the language). Below, we give the definitions of the main operators used in this paper.

$$\begin{aligned}
&\text{Collect} \quad \Gamma[x : f(x) \mid p(x)](e) && = \{f(x) \mid x \in e \wedge p(x)\} \\
&\text{Semijoin} \quad e_1 \underset{x_1,x_2:p(x_1,x_2)}{\ltimes} e_2 && = \{x_1 \mid x_1 \in e_1 \wedge \exists x_2 \in e_2 \bullet p(x_1,x_2)\} \\
&\text{Antijoin} \quad e_1 \underset{x_1,x_2:p(x_1,x_2)}{\triangleright} e_2 && = \{x_1 \mid x_1 \in e_1 \wedge \nexists x_2 \in e_2 \bullet p(x_1,x_2)\} \\
&\text{Nestjoin} \quad e_1 \underset{x_1,x_2:f(x_1,x_2)|p(x_1,x_2);a}{\Delta} e_2 && = \{x_1 +\!\!+ \langle a = X\rangle \mid x_1 \in e_1 \wedge X = \\
& && \qquad \{f(x_1,x_2) \mid x_2 \in e_2 \wedge p(x_1,x_2)\}\}
\end{aligned}$$

The collect operator Γ is simply a syntactic variant of the SELECT-FROM-WHERE construct of SQL:

$$\text{SELECT } f(x) \text{ FROM } x \text{ IN } e \text{ WHERE } p(x) \equiv \Gamma[x : f(x) \mid p(x)](e)$$

We have two special forms of Γ-expression:

1. $\Gamma[x : f \mid true](X) \equiv \alpha[x : f](X)$ which is also known as the map operator, and
2. $\Gamma[x : x \mid p](X) \equiv \sigma[x : p](X)$ which is also known as the selection operator.

The nestjoin operator was introduced in [17]. The operator is a combination of join and grouping and was introduced to avoid problems with dangling tuples. Parameters of the nestjoin operator are a predicate p, a function f, and a label a. Each tuple in the left-hand join operand is concatenated with the unary tuple $\langle a = X\rangle$; the set X consists of the right-hand operand tuples that satisfy p, modified according to function f.

In addition to the operators listed above, we have the standard set operators union, difference, and intersection, the tuple constructor $\langle a_1 = e_1, \ldots, a_n = e_n \rangle$, tuple projection $x[L]$, which is denoted as x_L, projection π, join $\bowtie$, nest ν and unnest μ, etc. The **except**-construct can be used as a shorthand for explicit tuple construction. Let t denote the unary tuple $\langle a = 1 \rangle$, then we have: t **except** $(a = 2, b = 3) = \langle a = 2, b = 3 \rangle$. The **except**-construct is used to modify attribute values, or to extend tuples with new attribute values. Predicates of the language may be arbitrary Boolean expressions, involving set comparison operators and quantifier expressions $\exists x \in X \bullet p$ and $\forall x \in X \bullet p$.

In this paper, capitals X, Y, Z are used to denote table expressions, i.e. base tables or set expressions with base table operands only. The expression $FV(e)$ stands for the set of free variables that occur in some expression e. Operators collect, select, map, and quantifiers $\exists$ and $\forall$ are called *iterators*.

3 Approach

In translation, the goal is to remove nested iterator occurrences as much as possible, by rewriting into efficient join expressions.

SQL languages offer the possibility to formulate nested queries, i.e. queries that contain subqueries (nested query blocks). In the relational model, a subquery operand is a base table or another subquery. In the translation to relational algebra, subqueries are removed by transforming nested queries into join queries. Transformation into join queries is advantageous because the join can be implemented such that its performance is better than pure nested-loop execution expressed by a nested query. Also for nested relational and object-oriented systems, the set-oriented paradigm seems appropriate. Though navigation has been considered as the prevailing method to access object-oriented databases in the past, recently more attention has been paid to set-oriented access methods, like pointer-based joins [16].

In a language such as OSQL, arbitrary nesting of query blocks may take place, in the SELECT- as well as in the WHERE-clause. The operands of nested query blocks may be base tables or set-valued attributes (or other subqueries); the two forms of iteration may alternate in arbitrary ways. The goal in translation is to achieve set-orientation, i.e. to remove nested iteration as much as possible. Nested iteration can be removed in two different ways:

Unnesting of Expressions Unnest rules may be applied either to the top level expression, moving nested base table occurrences to the top level, or to nested expressions, introducing nested set operations with base table and/or set-valued operands.

Unnesting of Attributes Set-valued attributes can be unnested using the operator μ, and nested later on, if necessary, using ν.

Depending on the expression concerned, one or more options may be appropriate. In this paper, we do not consider the option of attribute unnesting; it can be treated independently and easily incorporated into our transformation algorithm. We present some example equivalences:

Example 1 From Nested Expressions to Joins

1. $\sigma[x : \exists y \in Y \bullet x.a = y.a](X) \equiv X \underset{x,y:x.a=y.a}{\ltimes} Y$
2. $\alpha[x : \langle a = x.a, c = \Gamma[y : y.c \mid x.a = y.a](Y)\rangle](X) \equiv$
   $\alpha[v : \langle a = v.a, c = v.ys\rangle](X \underset{x,y:y.c|x.a=y.a;ys}{\Delta} Y)$
3. $\alpha[x : \Gamma[y : x.a + y.a \mid y.b = 1](Y)](X) \equiv$
   $\alpha[v : v.ys](X \underset{x,y:x.a+y.a|true;ys}{\Delta} \sigma[y : y.b = 1](Y))$

The left-hand side of the expressions above express a pure nested-loop execution strategy. By rewriting into join expressions, other implementation options come within reach.

We strive to achieve an efficient translation, i.e. to obtain logical algebra expressions that have reasonable performance. In our opinion, query optimization should not be restricted to the phases of algebraic rewriting and plan compilation; optimization should play a role in all phases of query processing. Simple, standard algorithms for translating the user language (either SQL or calculus-like) into the algebra may result in very inefficient expressions [19]. The inefficiency introduced in the translation phase is assumed to be reduced in the phase of logical optimization. This is quite a hard task, because in a pure algebraic context, in which information about the original query structure is scattered throughout the expression, it becomes difficult to find the proper optimization rules and to control the sequence of rule application [14]. To support this claim, we invite the reader to transform the expression:

$$((\sigma[x : p(x)](X) \times Y) \cup (X \underset{x,y:q(x,y)}{\bowtie} Y)) \div Y$$

into:

$$X - \sigma[x : \neg p(x)](X) \underset{x,y:\neg q(x,y)}{\ltimes} Y$$

using algebraic rewrite rules only. Both expressions are translations of the expression:

$$\sigma[x : \forall y \in Y \bullet p(x) \vee q(x,y)](X)$$

The first uses division to handle the universal quantifier, the second set difference. In our opinion, it is better to try to achieve a 'good' translation right away than to try to rewrite inefficient algebraic expressions afterwards.

4 Translation

In this section, we present our basic transformation rules. We restrict ourselves in that we discuss nested iteration only. We do not consider nested occurrences of set (comparison) operators, i.e., we leave them as they are.

The input to the transformation is a collect expression $\Gamma[x : f \mid p](e)$. Recall that the collect is the syntactic equivalent of the SQL SELECT-FROM-WHERE construct. The expressions f, p, and e may be arbitrary, containing other collects and/or quantifier expressions. Operands of nested iterators may be tables as well as set-valued attributes. The goal is to remove nested collects and quantifiers by rewriting into join operators. We want to achieve an efficient translation, i.e., we try to (1) avoid Cartesian products as much as possible, (2) push through predicates and functions, and (3) give preference to cheap operators, given a choice.

The basic rewrite algorithm consists of two steps: standardization and unnest. *Standardization* involves composition and predicate transformation. In *unnest*, subqueries are removed from parameter expressions by the introduction of join operators. The two steps are described in more detail below.

4.1 Standardization

Standardization involves composition and predicate transformation. In composition, collect operands and quantifier range expressions that are iterator expressions are transformed into table or attribute expressions; composition means the combination two iterators into one. As in the relational context, composition is needed because the user/system-prescribed order of operations (evaluation of predicates and functions) is not necessarily the most efficient. In addition, composition may offer additional optimization opportunities. We deal with the three iterators Γ, $\exists$, and $\forall$, therefore, we need the following rules:

Rule 1 Composition

1. $\Gamma[x : f(x) \mid p(x)](\Gamma[x : g(x) \mid q(x)](X)) \equiv \Gamma[x : f(g(x)) \mid p(g(x)) \wedge q(x)](X)$
   Note that the right-hand side contains the common subexpression $g(x)$.
2. $\exists x \in \Gamma[x : f(x) \mid p(x)](X) \bullet q(x) \equiv \exists x \in X \bullet p(x) \wedge q(f(x))$
3. $\forall x \in \Gamma[x : f(x) \mid p(x)](X) \bullet q(x) \equiv \forall x \in X \bullet \neg p(x) \vee q(f(x))$

The output of the phase of composition is a possibly nested collect expression in which the operand of each iterator is either a base table, a set-valued attribute, or a set expression (union, product), but not an iterator expression.

In the relational context, predicates usually are rewritten into Prenex Normal Form (PNF). After transformation into PNF, the matrix of the PNF expression can be optimized [11]. However, in [2], it is proposed to use a different normal form for predicates, namely the Miniscope Normal Form (MNF), in which quantifier scopes do not contain subexpressions that do not depend on the quantifier variable itself. Allegedly, MNF allows for a better translation, i.e. a translation with better results. As we will see, rewriting into MNF is one example of the generally beneficial rewrite strategy to remove local constants from iterator parameter expressions; therefore, we rewrite into MNF, according to the rules of [2].

4.2 Unnest

Below, we present a basic set of rewrite rules for the transformation of nested expressions into join or product expressions. We present rules that are generally valid: they can be applied to arbitrary subexpressions occurring at arbitrary levels. In Section 4.3 and Section 4.4, we discuss the restrictions placed on rule application and the rewrite strategy adhered to, respectively.

Whenever quantifiers occur in predicates between blocks, we may apply the rewrite rules presented below, translating nested expressions with quantification into relational join (product, join, semi-, or antijoin) operations. Existential quantification can be removed by the introduction of a semijoin, regular join or product operator:

Rule 2 Unnesting Existential Quantification

1. $\Gamma[x : f \mid \exists y \in Y \bullet p(x,y)](X) \equiv \Gamma[x : f \mid true](X \underset{x,y:p(x,y)}{\ltimes} Y)$
2. $\Gamma[x : f(x) \mid \exists y \in \sigma[y : p(x,y)](Y) \bullet q(x,y)](X) \equiv$
   $\Gamma[v : f(v_X) \mid q(v_X, v_Y)](X \underset{x,y:p(x,y)}{\bowtie} Y)$
3. $\Gamma[x : f(x) \mid \exists y \in Y \bullet p(x,y)](X) \equiv \Gamma[v : f(v_X) \mid p(v_X, v_Y)](X \times Y)$

Universal quantification can be removed by introducing an antijoin or a product and a division operator:

Rule 3 Unnesting Universal Quantification

1. $\Gamma[x : f \mid \forall y \in Y \bullet p(x,y)](X) \equiv \Gamma[x : f \mid true](X \underset{x,y:\neg p(x,y)}{\triangleright} Y)$
2. $\Gamma[x : f \mid \forall y \in Y \bullet p(x,y)](X) \equiv \Gamma[x : f \mid true]((\Gamma[v : v \mid p(v_X, v_Y)](X \times Y)) \div Y)$

For expressions that contain nested collect operators, we have the following equivalence rules (these are explained below):

Rule 4 Unnesting Collect

1. $\Gamma[x : E(x, \Gamma[y : f \mid p](Y))](X) \equiv \Gamma[v : E(v_X, v.ys)](X \underset{x,y:f|p;ys}{\Delta} Y)$
2. $\Gamma[x : E(x,Y)](X) \equiv \Gamma[v : E(v_X, v.ys)](X \underset{ys}{\Delta} Y)$

In our language, we have at our disposal the complex object equivalent of the relational Cartesian product, namely the *nested Cartesian product*, which consists of a nestjoin operator with join predicate *true* and nestjoin function identity:

$$X \underset{x,y:y|true;a}{\Delta} Y \equiv X \times \{\langle a = Y \rangle\}$$

(A product expression $X \Delta_{x,y:y|true;a} Y$ may be abbreviated as $X \Delta_a Y$.) Consider Rule 4(2) above. The left-hand side of this rule is a collect, of which parameter expression E (an abbreviation of some expression $f \mid p$) contains a subexpression Y, which may be a base table, a subquery, a set expression, etc. Y may be removed from E by the introduction of the nested product $X \Delta_{ys} Y$. In the collect expression, now having as operand the nested product instead of table X, expression Y is replaced by the newly formed set-valued attribute ys. In addition, the occurrences of outer loop variable x must be adapted to account for the fact that the outer loop no longer iterates over X, but over the nested product, which has an additional attribute. The expression v_X delivers the original tuple value of x. Note that $v_X.a$, with a an arbitrary label, is equivalent to $v.a$. We give an example:

Example 2 Nested Product

$$\Gamma[x : x \mid x.c \subseteq \sigma[y : x.a = y.a](Y)](X) \equiv$$
$$\Gamma[v : v_X \mid v.c \subseteq \sigma[y : x.a = y.a](v.ys)](X \underset{ys}{\Delta} Y)$$

So any (closed) nested table expression can be moved to the top level by the introduction of a nested Cartesian product. A nested product expression can be looked upon as the equivalent of the relational project-select-product expression. Like its relational equivalent, it is highly inefficient; it can be considered as the Most Costly Normal Form [12]. A nested product expression can be optimized by pushing through predicates and functions. However, we choose for a better translation rule. Consider Rule 4(1), the left-hand

side of which denotes a collect that contains a nested collect expression. Nested collect expressions can be removed from collect parameter expressions (and from quantifier scopes as well) by the introduction of a (proper) nestjoin instead of a nested product. We give an example of this rule:

Example 3 Nestjoin

$$\Gamma[x : x \mid x.c \subseteq \sigma[y : x.a = y.a](Y)](X) \equiv \Gamma[v : v_X \mid v.c \subseteq v.ys](X \underset{x,y:y \mid x.a=y.a;ys}{\Delta} Y)$$

4.3 Restrictions on Rule Application

The rules given above may be applied at will: at nested levels, to expressions concerning base table as well as set-valued attribute operands, etc. Theoretically, the only restriction on rule application is that it is not allowed to introduce free variables—specifically, it holds that the left join variable may not occur free in the right hand join operand. Also, whenever an unnesting rule is applied at top level, then it must hold that predicates and functions do not contain free variables other than the (nest)join variables themselves; if a rule is applied at nested levels, variables from higher levels may occur free in function and/or predicate. However, in practice we require that join predicates are closed dyadic formulas. As an exception, a nestjoin predicate may be missing (equivalent to true), whenever a dyadic nestjoin function is present.

Also, in theory, join predicates may be arbitrary expressions. Join predicates may contain set operators, iterators, as well as base table occurrences. For example, it is perfectly legal to write $X \bowtie_{x,y:x.a \in y.c} Y$, or even $X \bowtie_{x,y:x.a \in \Gamma[z:z \mid p(x,y,z)](Z)} Y$. Though work is being done on the efficient implementation of joins with complex join predicates, e.g. [9, 10], present join implementation techniques are not capable of handling complex join predicates; these probably will be handled by nested-loop execution after all. We therefore require that join predicates consist of atomic terms only. Atomic terms are comparisons between attribute values and/or constants of atomic type. Note that, consequently, join predicates do not contain base table occurrences.

Finally, in theory nestjoin functions may be arbitrary expressions as well. However, we require that nestjoin functions do not contain base table occurrences, precisely because the purpose of unnesting is to move base tables occurrences to the top level.

So, in practice we require that (1) join predicates are closed dyadic formulas that consist of atomic terms only, (2) nestjoin functions do not contain base table occurrences.

4.4 Rewrite Strategy

We propose the following initial rewrite strategy. Whenever possible, we push through predicates and functions to joins and join operands. Rules for pushing through predicates to regular joins and join operands can be found for example in [7]. In an extended version of this paper [19], we present rules for pushing through operations (predicates and functions) to nestjoin operands.

In unnesting, we use a top-down strategy, recursively joining outermost iterator operand with next inner, if possible. We first consider pairs of table expressions, then pairs

of table expressions and set-valued attributes, and finally pairs of set-valued attributes. In other words, first we try to join the top level operand, which is a table, with each of the nested base table occurrences in the query, in order of nesting level. Next, we take the next inner iterator operand that is a base table, and follow the same procedure. After having considered each pair of base tables, we consider joining pairs of tables and set-valued attributes, and finally pairs of set-valued attributes. In case there is a choice, with multiple subqueries, the order of unnesting is arbitrary. We prefer partial joins (semi- and antijoin) to the regular join, and the regular join to the nestjoin. In the first instance, we avoid Cartesian products; these are introduced only if everything else has failed. The advantage of a top-down unnesting strategy is that the level of nesting in the algebraic expression is kept to a minimum. We give an example. Assume we have the following query:

$$\alpha[x : \Gamma[y : \sigma[z : x.a = z.a \wedge y.b = z.b](Z) \mid x.a = y.a](Y)](X)$$

or, equivalently:

$$\alpha[x : \alpha[y : \sigma[z : y.b = z.b](\sigma[z : x.c = z.c](Z))(\sigma[y : x.a = y.a](Y))](X)$$

We first join X and Y. Next, Z is joined with the top level nestjoin result, and finally, we introduce a local join between set-valued attributes ys and zs. The full rewriting is shown below:

Rewriting Example 1

$$\begin{aligned}
&\alpha[x : \alpha[y : \sigma[z : y.b = z.b](\sigma[z : x.a = z.a](Z))(\sigma[y : x.a = y.a](Y))](X) \\
&\equiv \alpha[v : \alpha[y : \sigma[z : y.b = z.b](\sigma[z : v.a = z.a](Z))(v.ys)](X \underset{x,y:y[x.a=y.a;ys}{\Delta} Y) \\
&\equiv \alpha[w : \alpha[y : \sigma[z : y.b = z.b](w.zs)](v.ys)](X \underset{x,y:y|x.a=y.a;ys}{\Delta} Y) \underset{v,z:z|v.a=z.a;zs}{\Delta} Z) \\
&\equiv \alpha[w : \alpha[t : t.yzs](v.ys \underset{y,z:z|y.b=z.b;yzs}{\Delta} w.zs)](X \underset{x,y:y|x.a=y.a;ys}{\Delta} Y) \underset{v,z:z|v.a=z.a;zs}{\Delta} Z)
\end{aligned}$$

Whenever the operands of nested iterators are set-valued attributes instead of base tables, the result may contain nested joins.

5 Splitting Expressions

Given our set of rewrite rules presented above for unnesting quantifiers and collects (together with rules for pushing through operations to join (operands)), and given our restrictions concerning join predicates, often the only option will be to introduce (nested) Cartesian products, because predicates and functions do not have the right format. For example, consider the expression:

$$\Gamma[x : \Gamma[y : y \mid p(x,y) \wedge \exists z \in Z \bullet q(x,z) \wedge r(y,z)](Y) \mid true](X)$$

The nested collect predicate is not atomic, so we cannot use the rule for unnesting a collect. Also, the quantifier cannot be removed because it does not occur at the top level. Before we can apply the unnesting rules, the predicate has to be rewritten. Splitting of expressions enables us to introduce joins instead of Cartesian products. Below, we present equivalence rules used in splitting.

5.1 Equivalence Rules

Rules for splitting predicates are the following:

Rule 5 Splitting Predicates

1. $\Gamma[x : f \mid p \wedge q](X) \equiv\equiv \Gamma[x : f \mid q](\sigma[x : p](X))$
2. $\Gamma[x : f \mid p \vee q](X) \equiv \Gamma[x : f \mid p](X) \cup \Gamma[x : f \mid q](X)$
3. $\exists x \in X \bullet p \wedge q \equiv \exists x \in \sigma[x : p](X) \bullet q$
4. $\forall x \in X \bullet p \vee q \equiv \forall x \in \sigma[x : \neg p](X) \bullet q$

Expressions that denote function results can be split of as well. Let Θ denote either Γ, $\forall$, or $\exists$. For reasons of convenience, in the rules given below quantification $\forall x \in X \bullet p$ and $\exists x \in X \bullet p$ is written as $\forall[x : p](X)$ and $\exists[x : p](X)$, respectively. Expression $E(s_1, \ldots, s_n)$ denotes an iterator parameter expression E with subexpressions $s_1, \ldots, s_n$.

Rule 6 Splitting Functions

1. Let x not occur free in E outside of g, then:
   $\Theta[x : E(g(x))](X) \equiv \Theta[x : E(x)](\alpha[x : g(x)](X))$
2. Let x not occur free in E outside of instances of the form $g_i(x)$, then:
   $\Theta[x : E(g_1(x), \ldots, g_n(x))](X) \equiv$
   $\Theta[x : E(x.a_1, \ldots, x.a_n)](\alpha[x : \langle a_1 = g_1(x), \ldots a_n = g_n(x)\rangle](X))$
3. Let X be a table, and let label a not occur in the schema of X, then:
   $\Theta[x : E(x, g(x))](X) \equiv \Theta[v : E(v_X, x.a)](\alpha[x : x \textbf{ except } (a = g(x))](X))$

We give an example illustrating each of the above rules:

Example 4 Splitting Functions

1. $\sigma[x : \exists y \in Y \bullet x.a = y.b + y.c](X) \equiv \sigma[x : \exists y \in \alpha[y : y.b + y.c](Y) \bullet x.a = y](X)$
2. $\sigma[x : \exists y \in Y \bullet x.a = y.a \wedge x.b = \text{COUNT}(y.c)](X) \equiv$
   $\sigma[x : \exists y \in \alpha[y : \langle a = y.a, b = \text{COUNT}(y.c)\rangle](Y) \bullet x.a = y.a \wedge x.b = y.b](X)$
3. $\sigma[x : \forall y \in Y \bullet \text{COUNT}(x.c) > y.a](X) \equiv$
   $\Gamma[v : v_X \mid \forall y \in Y \bullet v.a > y.a](\alpha[x : x \textbf{ except } (a = \text{COUNT}(x.c))](X))$

The syntactic form of the unnest rules, together with the restrictions posed on their application requires that predicates and functions are split to be able to introduce joins instead of products. Given our rewrite strategy and unnest rules, the way predicates are split determines the form of the result, w.r.t. join order and type of join operators present. We need heuristics to guide the splitting of predicates and functions. Below, we present some of the heuristic rules that can be used to achieve a better translation.

5.2 Heuristics

Given a choice, expressions are split such that the cheapest (most restrictive predicate, less costly function) expression part is evaluated first. Iterator parameter expressions can be classified according to the number of variables that occurs free (constant, monadic, dyadic, multi-variable), and the presence of other iterators (simple versus complex). The following nested expression will serve as the leading example in this section:

$$\sigma[x : \exists y \in Y \bullet \exists z \in Z \bullet p_1(x) \wedge p_2(y) \wedge p_3(x, y) \wedge p_4(z) \wedge p_5(x, z) \wedge p_6(y, z)](X)$$

We remark that we have chosen the above example just for illustration purposes—it looks relational, but the same nesting pattern can be achieved with collects instead of quantifiers. Assume that X, Y, and Z are base tables and that all predicates p_i are atomic.

We note that the selection predicate is in PNF, and that the matrix of the PNF expressions contains conjuncts that do not depend on one or both quantifier variables. The predicate therefore is rewritten into MNF, as proposed in [2]:

$$\sigma[x : p_1(x) \wedge \exists y \in Y \bullet p_2(y) \wedge p_3(x,y) \wedge \exists z \in Z \bullet p_4(z) \wedge p_5(x,z) \wedge p_6(y,z)](X)$$

A comparable transformation that concerns a map operator is the following:

$$\alpha[x : \alpha[y : x.b + y.b](Y)](X) \equiv \alpha[x : \alpha[y : x + y.b](Y)](\alpha[x : x.b](X)) \text{ (Rule 6(1))}$$

The subexpression $x.b$ does not depend on the inner map variable y, so it can be evaluated outside of the scope of y. The above transformation corresponds to the idea of pushing through a projection. Even if attribute names occur at different nesting levels, projections can be pushed through:

$$\alpha[x : \langle a = x.a, c = \alpha[y : x.b + y.b](Y)\rangle](X) \equiv \alpha[v : \langle a = v.a, c = \alpha[y : v.b + y.b](Y)\rangle](\alpha[x : \langle a = x.a, b = x.b\rangle](X)) \text{ (Rule 6(2))}$$

We see that in a complex object model, as in the relational model, it is only necessary to preserve those attributes that are needed in subsequent computations. Subexpressions of parameter expressions that are constant w.r.t. the corresponding iterator variable, i.e. subexpressions in which the iterator variable does not occur free are called *local constants*. We have a first heuristic transformation rule:

Heuristic Rule 1 Local constants are removed from iterator parameter expressions as much as possible.

To remove independent subformulas from predicates we use the descoping rules (possibly others are needed too, see[2]):

Rule 7 Descoping Let $x \notin FV(p)$, then:

1. $\exists x \in X \bullet p \wedge q(x,y) \equiv p \wedge \exists x \in X \bullet q(x,y)$
2. $\exists x \in X \bullet p \vee q(x,y) \equiv p \vee \exists x \in X \bullet q(x,y)$[1]

To obtain independent subformulas, the technique of quantifier exchange may be of help. For functions, we use the rules for splitting of functions as given in Rule 6. Note that w.r.t. quantifier scopes, independent subformulas can be removed completely. W.r.t. functions, this is not possible—in the above map transformation, the inner map still contains the local constant x.

Second, another type of constant expression is one in which no variables from higher levels occur free; this type of expression is called an *global constant*. Global constants are evaluated independently, which becomes possible by naming them by means of a local definition facility:

[1] Because variables are range-restricted, we have to take into account the possibility of empty ranges. The correct transformation is:

$$\textbf{if } X = \emptyset \textbf{ then } \mathit{false} \textbf{ else } p \vee \exists x \in X \bullet q(x,y)$$

For reasons of simplicity, we assume quantifier ranges are never empty.

$$\sigma[x : \exists y \in \sigma[y : p(y)](Y) \bullet q(x,y)](X) \equiv$$
$$\sigma[x : \exists y \in Y' \bullet q(x,y)](X) \textbf{ with } Y' = \sigma[y : p(y)](Y)$$

Heuristic Rule 2 Global constants are named with a local definition facility.

We return to our leading example. We notice two monadic conjuncts at the top level, i.e. $p_1(x)$ and the quantifier expression itself, and also two at a nested level, namely $p_2(y)$, and $p_4(z)$. Monadic expressions often can be pushed through to the corresponding iterator operand, thereby possibly creating global constants that can be evaluated independently:

$$\sigma[x : \exists y \in \sigma[y : p_2(y)](Y) \bullet p_3(x,y) \wedge$$
$$\exists z \in \sigma[z : p_4(z)](Z) \bullet p_5(x,z) \wedge p_6(y,z)](\sigma[x : p_1(x)](X)) \text{ (Rule 5)}$$

Heuristic Rule 3 Monadic expressions are evaluated first, if possible.

An example of Heuristic Rule 3 that concerns the map operator is the following:

$$\alpha[x : \alpha[y : x.b + y.b](Y)](X) \equiv \alpha[x : \alpha[y : x.b + y](\alpha[y : y.b](Y))](X) \text{ (Rule 6(1))}$$

Again, the above example deals with pushing through a projection. Notice however, that also very complex functions can be pushed through.

As mentioned, both p_1 as well as the top-level quantifier expression are monadic predicates; we made the assumption that predicate p_1 is atomic. Given a choice, it seems advantageous to evaluate expressions that do not contain iteration before the ones that do, i.e. to evaluate expressions in order of their respective nesting level. Whenever the nesting level is the same, we may choose an arbitrary evaluation order, or invent some other heuristic rule.

Heuristic Rule 4 Expressions are evaluated in order of nesting level.

So far so good. We have discussed constant and monadic expressions, and now we have to decide what to do with dyadic and multi-variable parameter expressions. Joins are binary operators, so dyadic expressions are candidates for join predicates and functions. In our example expression, we notice dyadic conjuncts p_3, p_5, and p_6 that mutually link the base tables that occur in the query. It is tried to split multi-variable predicates and functions as needed, i.e., as prescribed by the unnesting strategy. Whenever we investigate the possibility of joining two tables A and B, we search for maximal closed expressions of the proper format that refer to attributes of A and B.

Heuristic Rule 5 Predicates and functions are split as prescribed by the unnesting strategy.

In the relational model, predicates involve atomic attribute comparisons only, and multi-variable selection predicates and also quantifier scopes can be split easily. In a complex object model, we may have arbitrary functions as well as predicates, and splitting of multi-variable expressions is not always possible (consider for example the expression $\sigma[z : x.a \cup z.a = y.a](Z)$).

Nested collects can be removed independent from the nesting level, but quantification can be removed only by joining adjacent operands, i.e., in which the one operand occurs nested immediately within the other (or within a conjunction of predicates). Because we prefer flat joins above nestjoins, before starting the top-down unnesting procedure, we first try to introduce (nested) flat joins between base table expressions as much as possible. It is required that the nested join expressions are closed, to be able to move them to the top level. Given the example query:

$$\alpha[x : \sigma[y : p(x,y) \wedge \exists z \in Z \bullet q(x,z) \wedge r(y,z)](Y)](X)$$

we note an existential quantifier nested within a selection. Therefore, we split the multi-variable quantifier scope as needed, and first introduce a nested join between Y and Z:

Rewriting Example 2

$$\begin{aligned}
&\alpha[x : \sigma[y : p(x,y) \wedge \exists z \in Z \bullet q(x,z) \wedge r(y,z)](Y)](X) \\
&\equiv \alpha[x : \sigma[y : p(x,y) \wedge \exists z \in \sigma[z : r(y,z)](Z) \bullet q(x,z)](Y)](X) \text{ (Rule 5(3))} \\
&\equiv \alpha[x : \sigma[v : p(x,v_Y) \wedge q(x,v_Z)](Y \underset{y,z:r(y,z)}{\bowtie} Z)](X) \text{ (Rule 2(2))}
\end{aligned}$$

Heuristic Rule 6 Before starting the top-down unnesting procedure, we introduce closed flat join expressions at nested levels, if possible.

We now summarize the foregoing:

- Whenever possible, global constants are named, local constants are removed, and monadic subexpressions are pushed through, in order of nesting level. In addition, predicates and functions are pushed through to joins and join operands, whenever possible.
- We unnest, top down. We first join tables, then tables and set-valued attributes, and finally set-valued attributes. However, to handle nested quantification properly, we first try to introduce closed flat (nested) join operations. To enable the introduction of real joins instead of Cartesian products, parameter expressions are split as needed.

The above rewrite strategy is a starting point, i.e., in many cases it may be better to modify the strategy in one way or another.

5.3 Discussion

Traditionally, determination of the join order is done cost-based. We stress that in a complex object model with set-valued attributes that supports an algebra that contains a rich variety of (join) operators such a cost-based optimization may be hard to do. First, the proper algebraic equivalence rules have to be found. Second, the process of algebraic rewriting is difficult to control. By means of heuristic translation rules we try to achieve an algebraic expression that has performance that is not too bad right from the start.

The search space for an optimal join order is restricted. First, links between tables may be missing—we do not want to introduce Cartesian products. Second, iterator operands may be tables, as well as set-valued attributes, which cannot be moved to the top level. Whenever an iteration with an attribute operand is cast between iterators with

table operands, the result may contain nested joins. Also, the links between tables may be of a different nature, i.e., predicate and/or function, and the creation of predicate links between operands may be preferable to the establishment of function links. It is not clear what constitutes an optimal join order, from the viewpoint of logical optimization, i.e. not taking into consideration physical database characteristics. Generally speaking, it seems reasonable to assume that:

- Set operators are better than iterator expressions.
- Partial joins (semi- and antijoin) are better than regular joins, which in turn are better than Cartesian products. A flat join is better than a nestjoin.[2] Also, predicate links are better than function links, and atomic links are better than complex ones.
- Top-level operations are better than nested ones.
- Table operations are better than operations on set-valued attributes.

But is a nested semijoin better or worse than a top-level join? We remark that it is not always easy to judge expressions on relative performance without the use of a more or less detailed cost model. For example, returning to our example, assume that operand Y is not a base table, but the set-valued attribute c of table X. We simplify our example:

$$\sigma[x : \exists y \in x.c \bullet p_3(x,y) \wedge \exists z \in Z \bullet p_5(x,z) \wedge p_6(y,z)](X)$$

The expression contains predicates that mutually link all three iterator operands. Existential quantifiers may be exchanged to move the attribute iterator inside, but this is not a generally valid strategy. We may choose to join X with Z at the top level, and then to execute a nested quantification:

$$\sigma[v : \exists y \in x.c \bullet p_3(v_X, y) \wedge p_6(y, v_Z)](X \underset{x,z:p_5(x,z)}{\bowtie} Z)$$

Also, it is possible to join table Z with attribute c at a nested level:

$$\sigma[x : \exists v \in (x.c \underset{y,z:p_6(y,z)}{\bowtie} Z) \bullet p_3(x, v_Y) \wedge p_5(x, v_Z)](X)$$

The former is likely to be better than the second, because in the second tuples of Z are replicated for each matching tuple in attribute c, for each (set) value c. However, the performance of both expressions depends on join methods used, join selectivities, the respective cardinalities of join operands, etc.

6 Related Work

The work presented here follows that done on the translation and optimization of relational SQL. Basically, there are two ways of optimizing SQL: (1) the rewriting of SQL expressions themselves [13], and (2) translation of SQL into relational algebra [4], followed by algebraic rewriting. The underlying idea is that nested-loop expressions should be transformed into set expressions that do not contain nested operators. The important difference with the work presented here is that (1) we have to deal with nesting in the SELECT-clause, which is not allowed in relational SQL, and (2) the presence of set-valued attributes, which is non-relational as well. SQL languages proposed for complex

[2] This is questionable: a nestjoin does not suffer from data replication, but is not commutative like the regular join.

object models usually are orthogonal languages; the problem of choosing an algebra for, and of translation into the algebra of such a language is much more complicated.

The work presented in [5] and [6] has much in common with ours. In [6] a binary grouping operator is defined that differs slightly from the nestjoin in the sense that the join function is applied to the *set* of matching right-hand operand tuples, not to the elements themselves. This has as a consequence that nested expressions with so-called projection dependency, which involve a free variable occurring in a SELECT-clause, cannot be unnested. Generally speaking, the algebraic operators and the (algebraic) equivalence rules of [6] are similar to ours. However, in this paper, we consider queries with arbitrary nesting levels, incorporating nested iterators with set-valued attribute operands.

In [17], we showed that in complex object models, the regular, i.e. flat relational join operator does not suffice for the unnesting of nested queries. To solve this problem, we introduced the nestjoin. In [18], we presented a general strategy for unnesting. We proposed to use relational (join) operators whenever possible, and to use the nestjoin otherwise. In this paper, the general strategy outlined in [18] is made mode concrete. Starting from a basic set of transformation rules, we have discussed what can be done to achieve efficient algebraic expressions. We use heuristic rules to determine an initial join order, and to push through predicates and functions.

7 Conclusions and Future Research

In this paper, we have presented a heuristics-based, extensible algorithm for the translation of nested OSQL queries into efficient join expressions. Queries that involve nested quantifier expressions are translated using relational algebra operators. For the translation of nested queries that cannot be translated into flat join queries, the nestjoin operator is used, which is a combination of join and grouping. During translation, predicates and functions are pushed through as far as possible. We have presented a general framework that can easily be extended.

The main problem in the translation of nested OSQL queries is to find a good unnesting strategy. We have proposed a top-down unnesting strategy that minimizes the nesting level in nestjoin expressions. Our goal is to rewrite nested expressions into algebraic expressions such that expensive operators (e.g. Cartesian products), nested base table occurrences, and nested joins are avoided as much as possible.

How to achieve the above goal in the best possible way is topic of further research. A (heuristic) cost model may be needed to guide the transformation of nested queries; such a cost model is much more complex than the heuristic model used in logical optimization in the relational context, that merely prescribes to push through selections and projections. Other topics of future research for example are how to deal with nested set (comparison) operators, how to implement nested joins, and how to deal with more complex join predicates.

References

1. Balsters, H., R.A. de By, and R. Zicari, "Typed Sets as a Basis for Object-Oriented Database Schemas," *Proceedings ECOOP*, Kaiserslautern, 1993.

2. Bry, F., "Towards an Efficient Evaluation of General Queries: Quantifier and Disjunction Processing Revisited," *Proceedings ACM SIGMOD*, Portland, Oregon, June 1989, pp. 193–204.
3. R.G.G. Cattell, ed., *The Object Database Standard: ODMG-93*, Morgan Kaufmann Publishers, San Mateo, California, 1993.
4. 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, pp. 324–345.
5. Cluet, S. and G. Moerkotte, "Nested Queries in Object Bases," *Proceedings Fourth International Workshop on Database Programming languages*, New York, Sept. 1993.
6. Cluet, S. and G. Moerkotte, "Classification and Optimization of Nested Queries in Object Bases," manuscript, 1994.
7. Elmasri, R. and S.B. Navathe, *Fundamentals of Database Systems*, Benjamin/Cummings Publishing Company Inc., 1989.
8. Graefe, G., "Query Evaluation Techniques for Large Databases," *ACM Computing Surveys*, 25(2), June 1993, pp. 73–170.
9. Hellerstein, J.M. and A. Pfeffer, "The RD-Tree: An Index Structure for Sets," Technical Report #1252, University at Wisconsin at Madison, October 1994.
10. Ishikawa, Y., H. Kitagawa, and N. Ohbo, "Evaluation of Signature Files as Set Access Facilities in OODBs," *Proceedings ACM SIGMOD*, 1993, pp. 247–256.
11. Jarke, M. and J. Koch, "Query Optimization in Database Systems," *ACM Computing Surveys*, 16(2), June 1984, pp. 111–152.
12. 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 Kaufmann Publishers, San Mateo, California, 1993.
13. Kim, W., "On Optimizing an SQL-like Nested Query," *ACM TODS*, 7(3), September 1982, pp. 443–469.
14. Nakano, R., "Translation with Optimization from Relational Calculus to Relational Algebra Having Aggregate Functions," *ACM TODS*, 15(4), December 1990, pp. 518–557.
15. Pistor, P. and F. Andersen, "Designing a Generalized NF^2 Model with an SQL-Type Language Interface," *Proceedings VLDB*, Kyoto, August 1986, pp. 278–285.
16. Shekita, E.J. and M.J. Carey, "A Performance Evaluation of Pointer-Based Joins," *Proceedings ACM SIGMOD*, Atlantic City, May 1990, pp. 300–311.
17. Steenhagen, H.J., P.M.G. Apers, and H.M. Blanken, "Optimization of Nested Queries in a Complex Object Model," *Proceedings EDBT*, Cambridge, March 1994, pp. 337–350.
18. Steenhagen, H.J., P.M.G. Apers, H.M. Blanken, and R.A. de By, "From Nested-Loop to Join Queries in OODB," *Proceedings VLDB*, Santiago de Chile, September 1994.
19. Steenhagen, H.J., *Optimization of Object Query Languages*, Ph.D. Thesis, University of Twente, October 1995.