Nested Queries in Object Bases

Sophie Cluet
INRIA, Domaine de Voluceau
78153 Le Chesnay Cedex FRANCE

Guido Moerkotte
Fakultät für Informatik, Universität Karlsruhe
D–7500 Karlsruhe Germany

Abstract

Many declarative query languages for object-oriented databases allow nested subqueries. This paper contains the first (to our knowledge) proposal to optimize them. A two-phase approach is used to optimize nested queries in the object-oriented context. The first phase—called *dependency-based optimization*—transforms queries at the query language level in order to treat common subexpressions and independent subqueries more efficiently. The transformed queries are translated to nested algebraic expressions. These entail nested loop evaluation which may be very inefficient. Hence, the second phase *unnests* nested algebraic expressions to allow for more efficient evaluation.

1 Introduction

Many declarative query languages for object-oriented database management systems have been proposed in the last few years (e.g. [3, 5, 2, 18, 14]). To express complex conditions, access nested structure, or produce nested results, an essential feature found in these languages is the nesting of queries, i.e., the embedding of a query into another query.

The optimization of object-oriented (oo) queries has been intensively studied using algebraic rewriting [4, 7, 23, 24, 25] or rewriting of path expressions [7, 13, 14]. However, in spite of the importance of nested queries, we do not know of any research on their optimization. Nested queries in the oo context are usually translated into nested algebraic expressions which are evaluated through rather inefficient nested loops.

To a lesser extent, relational query languages also feature nested queries and their optimization has been considered in that contex. However, nested queries in the object-oriented context raise specific problems which have never been addressed before. The purpose of the present paper is to fill this gap. On one hand, we adapt some optimizing techniques from the relational context. On the other hand, we introduce novel techniques more directly related to the oo perspective.

The optimization of nested queries in the oo context should clearly use the vast body of knowledge on optimizing nested relational queries. Departing from the evaluation of nested SQL queries through nested loops [1], Kim proposed transforming SQL nested queries at the SQL level [19]. The leading tread behind the proposed transformations is to convert nested queries into joins so that a standard optimizer can work efficiently. However, there exist some major differences between nested queries and joins that are, as stated in [12], the creation of duplicates and the way empty tables are handled. Indeed, these differences are at the bottom of most of the bugs subsequently detected in the original algorithms [10, 17, 6, 8, 11, 12]. To solve them and among other techniques, [8, 11] introduced outer-joins. While these algebraic operators are a nice solution to some of the above problems, they raise new issues. If a sequence of join operations can easily be reordered, joins and outer-joins do not commute that easily and one has to find new ways of dealing with multi-layered nested queries [8, 20, 22]. Finally, a unifying framework for different unnesting strategies was proposed in [21].

There exist some significant differences between nested queries in the oo and the relational context. First, the result of an oo nested query is not always flat. Second, nested queries in the relational context were restricted to occur in the selection clause and, to a limited extent, in the range clause (nested constant blocks when querying views). In the oo context, nested queries may appear in any clause: the result clause (typically **select**, or **retrieve**), the range clause (typically **from** or **range**), and the selection clause (typically **where**). This implies that we have to consider queries that do not match any of the types introduced for classifying nested relational queries [19]. Third, nested queries in the oo context do not always correspond to algebraic operations (e.g., method calls, path expressions), and we have to find an appropriate treatment for these nested expressions as well. While the above considerations seem to complicate the issues, there also exists one essential difference that will somehow simplify the problem: set-valued attributes can be represented explicitly. In particular, a consequence is that there is no need to introduce *null* values to represent unnested empty sets.

As mentioned above, some relational techniques are easily adapted to the oo context. For instance, the whole idea of using joins and outer-joins for unnesting nested queries will be successfully applied throughout the paper. However, the differences between the two models lead us to consider new optimization techniques. We propose a two-phase optimization. The first phase addresses a new challenge related to the third difference stated above. During this phase we apply *dependency-based optimization* which transforms queries by factoring out constant or locally constant nested queries as well as common subexpressions. Although not new in the oo context [7, 14, 16], this factorization is essential for finding a good evaluation strategy. Then, the resulting queries are translated in a straightforward manner into the nested algebra. The algebra we use is an extension of the GOM algebra [15, 16] and features some nice properties most oo algebras lack (e.g., associativity of join operations). The second phase—called *algebraic optimization*—exploits new opportunities brought by the possibility to represent non-atomic attributes. More specifically, the applied algebraic equivalences will make extensive use of a new powerful grouping operation. We will see that the problems implied by the first and second differences stated above are solved by a combination of the two phases.

The paper is organized as follows. The next section introduces the model and the language we will be using thorough the paper. Section 3 summarizes the dependency based optimization mainly by means of an example. Section 4 introduces the algebra. The algebraic equivalences used to unnest nested algebraic expressions are presented in Section 5 and applied to some representative nested queries. Section 6 concludes the paper.

2 Preliminaries

The data model we are working on is similar to the O_2[9], GOM [14] or Exodus [5] model. It features objects that have an identity, that are manipulated through user-defined methods, whose structures are complex and that belong to classes that may be refined into subclasses. Each class has an extent which is a set containing all the objects belonging to the class. The model also features complex values with no identity, that are manipulated by standard operators and do not belong to classes. Hence, there are no extents for them.

A toy schema defined on this data model is given below. It will be used in the sequel of the paper.

A Database Schema	
Class Employee type tuple Method BestSales(): set(Sale) Method SoldItems(): set(Item)	(name: tuple(first_name: string, last_name: string), address: tuple(num: string, street: string, city: string), sales: set(Sale), dept: Department, speciality: string, age: integer, children: set(tuple(name: string, age: integer)))
Class Manager type tuple inherits Employee	(manages: set(Department))
Class Sale type tuple Method amount(): integer	(description: set(tuple(item: Item, qty: integer)) date: tuple(day: integer, month: integer, year: integer), emp: Employee)
Class Item type tuple	(code: string, description: string, kind: string, price: real)
Class Department type tuple	(LocatedIn: Store, items: set(Item) emps: set(Employee), manager: Manager)
Class Store type tuple	(name: string, city: string,

The schema consists of classes, each of which has an extent, a type and some methods (although they do not always appear in the description). The type describes the structure of these objects and the methods their behaviour. Class *Manager* is a subclass of class *Employee*, i.e. it inherits its structure and methods, and the objects of Class *Manager* are contained in the extent of Class *Employee*. Class *Manager* has an added attribute which consists of a set of objects of Class *Department*. Although this is not compulsory, all classes are of a tuple type. Attributes are either atomic values (e.g., *age*), or objects (e.g., *dept*) or what we refer to as complex values (e.g., *name*).

Throughout the paper we will use the O_2SQL language [2] for the examples. However, the techniques that we present are not restricted to this language and can easily be applied to other languages as well. The O_2SQL language is functional. It is defined by a set of basic functions/queries and a way of building new ones through composition and iterators. There exist basic functions for querying persistent roots (e.g., *Company* is a query that returns the set of objects of Class *Company*), atomic values (e.g., arithmetical operations, comparison operations), tuple values (selection of an attribute), set values (e.g., flatten, avg) and objects (method calls). O_2SQL may directly access the values of the objects (violating encapsulation) when used interactively. The basic functions may be combined in any way, the only limitation being a correct typing. A function that expects an argument of a tuple type, for example, cannot be invoked with an argument of a set type. O_2SQL features several iterators: *select-from-where*, *exists*, *forall*, *group-by* and *sort*. In this paper, we will only consider the first.

As stated above, a method can be invoked inside a query. This raises many optimization issues (cost model, code optimization, side effects) that, to our knowledge, are still unsolved. Our goal here is not to study them. We do not present a cost model, do not study method code optimization and consider only methods without side effects. Thus, for us, a method is very much like a stored attribute.

3 Dependency-Based Optimization

In the relational context, a nested query is one containing a block nested inside. In the object-oriented context, things are different. Operations inside a block may be method calls, long path expressions, set operations on attributes, etc. Hence, we have to consider the optimization of nested expressions—which are also queries by definition—that are not blocks. We illustrate this with the following example:

```
select  tuple(name:emp.name,
              sale:emp_sale.description,
              month: emp_sale.date.month)
from    emp in Employee,
        emp_sale in Sale
where   emp_sale in emp.BestSales() and
        emp_sale.amount() > avg(select  sale.amount()
                                from    sale in Sale
                                where   sale.date.month = emp_sale.date.month)
```

The method call to *BestSales* is used in the join predicate. We consider this as a nested query. A good optimization will push it out of the join operation so that it won't be evaluated more than necessary. The *emp_sale.date.month* path expression in the nested block is also a nested query that should be pushed out of (i) its nesting block and (ii) the join performed in the higher level block.

In this paper, we consider two kinds of optimization of nested queries that are complementary. One is based on dependencies (treated in this section) and the other on algebraic equivalences (cf. Section 5). We will also take advantage of the dependency-based optimization to perform some common subexpression factorization (e.g., the path expression *emp_sale.date.month* in the previous example).

The dependency optimization is performed at the O_2SQL level. This kind of optimization, although vital, is simple enough and requires mainly one traversal of the syntax tree of the query. Since it had been presented elsewhere [7], we will not detail it. Let us just say that the algorithm consists in splitting every operation. The splitting allows us to consider every possible subexpressions that can be factorized. In this paper, we will not split everywhere but only when necessary in order to demonstrate the major points. We now present the transformed above O_2SQL query and comment on it. More examples can be found in Section 5.

```
select  tuple(name:en, sale:esd, month: esdm)
from    emp in Employee,
        emp_sale in ebs
where   esa > as
define  n = emp.name,
        esd = emp_sale.description,
        esdm = emp_sale.date.month,
        ebs = emp.BestSales(),
        esa = emp_sale.amount(),
        as = avg(select  sa
```

from	sale **in** Sale
where	sdm = esdm
define	sa = sale.amount()
	sdm = sale.date.month

The transformed query features a new **define** clause in each block. It is used to introduce variables representing expressions dependent only on its owner block. Note that the outer block **define** clause contains a variable for the *emp_sale.date.month* that is referenced twice (common subexpressions factorization): once in the inner block (constant subexpression factorization) and once in the outer block. During the translation to the algebra, we make use of the variable dependencies in order to push simple operations as far in as possible; especially before join operations are applied.

4 The Algebra

The core of the algebra consists of the following operators that are all defined on set values[1]: union ($\cup$), intersection ($\cap$), difference ($\setminus$), selection (σ), join($\bowtie$), left outer join ($⟕$), mapping (χ), d-join ($< \cdot >$), grouping (Γ). Except for d-join and grouping, these operators are rather standard. The d-join operation is used for performing a join between two sets, the second one being dependent on the first. It is left-associative and will be applied in post order. This operator can be used for unnesting and is, in many cases, equivalent to a join between two sets with a membership predicate [23]. We introduced it in order to cope with the values of types that do not have extension (i.e. there exist no sets on which a join could be applied). The grouping operator will make use of the fact that in the object-oriented context attributes can be set-valued. As we will see, this is useful for both unnesting nested queries and producing nested results.

Before defining the algebraic operators, we introduce some notations used in the sequel:

- We denote the type τ of an expression e by $e :: \tau$.
- The function $\mathcal{A}$ is defined as follows:
  $\mathcal{A}(e) = \{a_1, \ldots, a_n\}$ if $e :: \{[a_1 : \tau_1, \ldots, a_n : \tau_n]\}$ or $e :: [a_1 : \tau_1, \ldots, a_n : \tau_n]$
- $\mathcal{F}(e)$ denotes the set of all free variables of e.
- In order not to be explicit about the parameters of an expression, i.e., avoiding the λ notation, we introduce the following conventions:
  - For an expression e with free variables $\mathcal{F}(e) = \{a_1, \ldots, a_n\}$ and a tuple t with $\mathcal{F}(e) \subseteq \mathcal{A}(t)$ we define $e(t) = e[a_1 \leftarrow t.a_1, \ldots, a_n \leftarrow t.a_n]$. Similarily, we define $e(t_1, t_2)$ for binary operations. Note that the attribute names of t_1 and t_2 should be distinct in order to avoid name conflicts.
  - For an expression e with only one free variable x, we define $e(t) = e[x \leftarrow t]$.
- The symbol $\circ$ denotes function concatenation and (as a special case) tuple concatenation.
- Application of a function f to an argument e is denoted by either regular (e.g., $f(e)$) or dot (e.g., $e.f$) notation.
- As an abbreviation for a predicate $a_1\theta b_1, \ldots, a_n\theta b_n$ $(x.a_1\theta_1 y.b_1, \ldots, x.a_n\theta_n y.b_n)$ we often use $A\theta B$ $(x.[A]\theta y.[B])$ if $A = \{a_1, \ldots, a_n\}$ and $B = \{b_1, \ldots, b_n\}$.
- $\overline{A}(e)$ is used for denoting $\mathcal{A}(e) \setminus A$. When e is clear from the context, we use $\overline{A}$ as a shorthand.

We now define the operators needed in the sequel. Let $\mathcal{E}$ be the valuation of an expression. Then

$$
\begin{aligned}
\mathcal{E}(\chi_{e_2}(e_1)) &= \{\mathcal{E}(e_2(x)) | x \in \mathcal{E}(e_1)\} \\
\mathcal{E}(\chi_{a:e_2}(e_1)) &= \{y \circ [a : \mathcal{E}(e_2(y))] | y \in \mathcal{E}(e_1)\} \\
\mathcal{E}(\sigma_p(e)) &= \{x | x \in \mathcal{E}(e), \mathcal{E}(p(x))\} \\
\mathcal{E}(e_1 \bowtie_p e_2) &= \{y \circ x | y \in \mathcal{E}(e_1), x \in \mathcal{E}(e_2), p(y, x)\} \\
\mathcal{E}(e_1 ⟕_p^{g=c} e_2) &= \{y \circ x | y \in \mathcal{E}(e_1), x \in \mathcal{E}(e_2), p(y, x)\} \cup \\
& \quad \{y \circ z | y \in \mathcal{E}(e_1), \neg\exists x \in \mathcal{E}(e_2)\ p(y, x), \mathcal{A}(z) = \mathcal{A}(e_2), g \in \mathcal{A}(e_2),
\end{aligned}
$$

[1]Lists and Bags will be considered in another paper.

$$
\begin{array}{rcl}
 & & z.g = \mathcal{E}(c), \forall a \in \mathcal{A}(e_2)\ a \neq g \succ z.a = NULL\} \\
\mathcal{E}(e_1 < e_2 >) & = & \{y \circ x | y \in \mathcal{E}(e_1), x \in \mathcal{E}(e_2(y))\} \\
\mathcal{E}(\Gamma_{g;A\theta;f}(e)) & = & \{y.[A] \circ [g : G] | y \in \mathcal{E}(e), \\
 & & G = f(\{x | x \in \mathcal{E}(e), x.[A]\theta y.[A]\}\} \\
\mathcal{E}(flatten(e)) & = & \{y | x \in \mathcal{E}(e), y \in x\} \\
\mathcal{E}(max_{g;m;a\theta;f}(e)) & = & [m : max(\{x.a | x \in \mathcal{E}(e)\}), g : f(\{x | x \in \mathcal{E}(e), x.a\theta m\})]
\end{array}
$$

Note that, apart from the χ and *flatten* operations, all these operations are defined on sets of tuples. This guarantees some nice properties among which is the associativity of the join operations. Note also that the operators may take complex expressions in their subscript, therefore allowing nested algebraic expressions.

The $\chi_{a:e'}(e)$ operation actually is a shorthand for mapping with tuple concatenation. The Γ, *flatten* and *max* operations are mainly needed for optimization purposes, as we will see in the sequel, but do not add power to the algebra. Note that a *min* operation similar to the *max* operation can easily be defined. Sometimes the χ operation is equivalent to a simple projection or renaming. In that case, we will use π instead of χ.

In the sequel we will use the following abbreviations: $\Gamma_{g;A;f}$ for $\Gamma_{g;A=;f}$, $\Gamma_{g;A}$ for $\Gamma_{g;A;id}$, $\bowtie$ for $\bowtie_{true}$, and $e[a]$ for $\chi_{[a:x]}(e)$.

The algebra is defined on sets whereas most OODBMS also manipulate lists and bags. We believe that our approach can easily be extended by considering lists as set of tuples with an added positional attribute and bags as sets of tuples with an added key attribute.

5 Algebraic Optimization

In this section we use representative example queries to present some original unnesting algebraic rewritings of nested queries. We are not concerned with other rewritings (e.g., using joins to evaluate path expressions, evaluating disjunctive predicates through union operation, etc.) although the algebra supports them. For each query, we show the result of the source level transformation, the algebraic translation and the rewriting.

Nested Blocks Within the Select Clause

Nested queries in the select clause appear when a nested result is required. The O_2SQL query we chose to illustrate this case is given on the left, its transformed form on the right:

```
select  tuple(dept: d,                        select  tuple(dept: d, emps: des)
              emps: select  e                 from    d in Department
                    from    e in Employee     define  des = select  e
                    where   e.dept = d)                     from    e in Employee
from    d in Department                                     where   ed = d
                                                            define  ed = e.dept)
```

Translating this expression into the algebra yields

$$
\begin{array}{rcl}
q & \equiv & \chi_{[dept:d,emps:des]}(\chi_{des:e_2}(Department[d])) \\
e_2 & \equiv & \chi_e(\sigma_{ed=d}(\chi_{ed:e.dept}(Employee[e])))
\end{array}
$$

The e_2 nested algebraic expression represents the nested block. A block with one variable in the from clause is translated by (i) a map operation for constructing tuples whose single attribute represents the variable (e.g., $Department[d]$), (ii) a map operation for evaluating the expressions depending on the block (e.g., $\chi_{des:e_2}$), (iii) a selection when needed (e.g., $\sigma_{ed=d}$) and finally (iv) a map operation for building the final result (e.g., $\chi_{[dept:d,emps:des]}$). Of course all the tuple constructions implied by this translation do not have to be performed (e.g., $Department[d]$ can be interpreted as a scan on *Department* with variable d). Further, $\sigma_{ed=d}(\chi_{ed:e.dept}(Employee[e]))$ can be processed by a single physical operation or an index scan.

The problem with this direct translation is that there is just one underlying evaluation plan for nested expressions which corresponds to a nested loop. As shown in [19], this kind of evaluation can be very inefficient. Indeed, in this example, it implies several scans on the *Employee* extension. The idea to solve this problem is to unnest the expression by performing a grouping on the nested expression followed by a left outer-join. We use an outer-join in order to consider empty nested sets that would be lost by a simple join operation. The main reason for favoring the grouping/outer-join combination is the possibility of several interesting implementations for this combination where the straightforward nested loop evaluation is only one of them. Another good reason is the reordering possibilities it offers in case of a more complex query (i.e., more levels of nesting) [8, 20, 22, 21]. Hence, we apply the following equivalence which allows to unnest nested χ operations:

$$\begin{aligned}\chi_{g:f(\sigma_{A_1=A_2}(e_2))}(e_1) &= \pi_{\overline{A_2}}(e_1 \mathbin{\text{⟕}}^{g=f(\emptyset)}_{A_1=A_2}(\Gamma_{g;A_2;f}(e_2))))\\ &\quad \text{if } A_i \subseteq \mathcal{A}(e_i),\ \mathcal{F}(e_2)\cap\mathcal{A}(e_1)=\emptyset,\ A_1\cap A_2=\emptyset, g\notin\mathcal{A}(e_1)\cup\mathcal{A}(e_2)\end{aligned}$$

The superscript $g = f(\emptyset)$ is the default value given when there is no element in the result of the group operation which satisfies $A_1 = A_2$ for a given element of e_1.

Applying this equivalence to our example query yields:

$$q \equiv \chi_{[dept:d,emps:des]}(\pi_{\overline{\{e,ed\}}}(Department[d] \mathbin{\text{⟕}}^{des=\emptyset}_{d=ed}(\Gamma_{des;ed;\chi_e}(\chi_{ed:e.dept}Employee[e]))))$$

With this expression, many evaluation plans can be considered. For instance, the grouping can be evaluated using a sort operation, or, if there exists an index, by a simple index scan. Note that if we know that every department has at least one employee, we can skip the outer-join operation. Indeed, in this specific example, it is used solely to generate empty sets of employees related to departments having no employee.

Nested Aggregated Block Within the Select Clause

Nested aggregated queries differ from the previous ones by the fact that, even though a grouping is required, the result will not be nested. To illustrate this case, we chose an example in which we rely on some semantic information to derive the fact that inner and outer queries have a common range. This is a rather common case when aggregate functions are involved. The next query is:

```
select  tuple(age:e1.age,
              nb:count(select  s
                       from    e2 in Employee
                               s in e2.sales
                       where   e2.age ≤ e1.age))
from    e1 in Employee
```

```
select  tuple(age:e1a, nb:cs)
from    e1 in Employee
define  e1a = e1.age,
        cs =  count(select  s
                    from    e2 in Employee
                            s in e2.sales
                    where   e2a ≤ e1a
                    define  e2a = e2.age)
```

and in the algebra

$$\begin{aligned} q &\equiv \chi_{[age:e1a,nb:cs]}(\chi_{cs:e_2}(e_1))\\ e_1 &\equiv \chi_{e1a:e1.age}(Employee[e1])\\ e_2 &\equiv count(\chi_s(\sigma_{e2a\le e1a}(e_2')))\\ e_2' &\equiv (\chi_{e2a:e2.age}Employee[e2]) < e2.sales[s] > \end{aligned}$$

The nested block represented by e_2 and e_2' features two variables in the from clause. This is translated by a d-join operation. Note that, in this block, we used informations on variable dependencies during the translation process to push the map operation on *Employee* before the d-join operation.

The main difference between this query and the previous one lies in the fact that $\le$ is used for comparison instead of "=". The resulting problem is that values occurring in e_1 might show up several times for one value of e_2.

Assume for a moment that every employee has at least one sale. Then, we may use advantageously the fact that one variable (e2) of the inner block ranges over the same set (all the employees) than the variable of the outer block (e1). Hence, we have:

$$\begin{aligned}\chi_{g:f(\sigma_{A_1\theta A_2}(e_2))}(e_1) &= \pi_{A1:A2}(\Gamma_{g;\theta A_2;f}(e_2))\\ &\quad if\ A_i\subseteq\mathcal{A}(e_i),\\ &\qquad \mathcal{F}(e_2)\cap\mathcal{A}(e_1)=\emptyset,\end{aligned}$$

$$e_1 = \pi_{A_1:A_2}(e_2) \quad (implies\ A_1 = \mathcal{A}(e_1))$$
$$g \notin \mathcal{A}(e_1) \cup \mathcal{A}(e_2)$$

By applying this equivalence to our example query, we get:

$$q \equiv \chi_{[age:e1a,nb:cs]}(\pi_{e1a:e2a}\Gamma_{cs;age\leq;count\circ\chi_s}(e_2'))$$

Note that the Γ operation can be efficiently implemented by first sorting the argument on its *age* values. If the argument is already sorted, e.g., due to an index scan, the Γ operation can be evaluated in linear time during a single scan of e_2, if f is linear.

Nested Queries With Flatten

Nested queries with flatten are commonly used in order to flatten some nested attributes. To discuss this case, consider the following example query:

```
flatten(select
          select tuple(name:c.name,age:c.age)
          from   c in e.children
          where  c.age < 18)
        from e in Employee)
```

```
flatten(select g
        from   e in Employee
        define ec = e.children
               g = select tuple(name:n,age:a)
                   from   c in ec
                   where  a < 18
                   define n = c.name
                          a = c.age )
```

The standard translation is

$$\begin{array}{rcl} q & \equiv & flatten(\chi_g(\chi_{g:e_2}(\chi_{ec:e.children}(Employee[e])))) \\ e_2 & \equiv & \chi_{[name:n,age:a]}(\sigma_{a<18}(\chi_{a:c.age,n:c.name}(ec[c]))) \end{array}$$

In order to push the flatten operation inside, we have to eliminate the redundant tuple extension for the attribute g:

$$\begin{array}{rcl} q & \equiv & flatten(\chi_{e_2}(\chi_{ec:e.children}(Employee[e]))) \\ e_2 & \equiv & \chi_{[name:n,age:a]}(\sigma_{a<18}(\chi_{a:c.age,n:c.name}(ec[c]))) \end{array}$$

Now, we know that for linear $f : \{\tau\} \rightarrow \{\tau'\}$ that

$$flatten(\chi_f(e)) \quad = \quad f(flatten(e))$$

Hence,

$$\begin{array}{rcl} q & \equiv & \chi_{[name:n,age:a]}(flatten(\chi_{e_2'})(\chi_{ec:e.children}Employee[e])) \\ e_2' & \equiv & \sigma_{a<18}(\chi_{a:c.age,n:c.name}(ec[c])) \\ q & \equiv & \chi_{[name:n,age:a]}(\sigma_{a<18}(flatten(\chi_{e_2''}(\chi_{ec:e.children}Employee[e])))) \\ e_2'' & \equiv & \chi_{age:c.age,n:c.name}(ec[c]) \\ q & \equiv & \chi_{[name:n,age:a]}(\sigma_{a<18}(\chi_{a:c.age,n:c.name}(flatten(\chi_{ec[c]}(\chi_{ec:e.children}Employee[e]))))) \\ & \equiv & \chi_{[name:n,age:a]}(\sigma_{a<18}(\chi_{a:c.age,n:c.name}(flatten(\chi_{e.children[c]}(Employee[e]))))) \\ & \equiv & \chi_{[name:n,age:a]}(\sigma_{a<18}(\chi_{a:c.age,n:c.name}(flatten(\chi_{children[c]}(Employee))))) \\ & \equiv & \sigma_{age<18}(\chi_{[name:n,age:a]}(\chi_{[a:age,n:name]}(flatten(\chi_{children}(Employee))))) \\ & \equiv & \sigma_{age<18}(\chi_{[name:name,age:age]}(flatten(\chi_{children}(Employee)))) \end{array}$$

where redundant tuple constructions where eliminated in the last steps. Note that the flatten operation is now applied on stored data. If there exists an appropriate index it can be evaluated very efficiently.

5.1 Nested Block Within the From Clause

As we will see, nested blocks in the from clause do not require nested algebraic expressions. They are queries whose original algebraic translation features operations pushed before a join (or d-join). Note that this kind of queries also occur in the relational context when views are queried. However, we do not know of any relational work where they are treated as simply as they should. The example is the following:

```
select  e
from    d in select  d
             from    d in Department
             where   d.locatedIn.city
                     == "Paris",
        e in Employee
where   e.dept == d
```

```
define DD = select  dd
            from    dd in Department
            where   dl == "Paris"
            define  dl = dd.locatedIn.city
select  e
from    d in DD,
        e in Employee
where   ed == d
define  ed = e.dept
```

The translation results in

$$
\begin{aligned}
q &\equiv \chi_e(\sigma_{ed=d}(\chi_{ed:e.dept}(e_1)))\\
e_1 &\equiv (DD[d] < Employee[d] >)\\
DD &\equiv \chi_{dd}(\sigma_{dl=\text{``}Paris\text{''}}(\chi_{dl:dd.locatedIn.city}(Department[dd])))\\
e_1 &\equiv (\pi_d(\sigma_{dl=\text{``}Paris\text{''}}(\chi_{dl:d.locatedIn.city}(Department[d])))) < Employee[e] >\\
&\equiv (\pi_d(\sigma_{dl=\text{``}Paris\text{''}}(\chi_{dl:d.locatedIn.city}(Department[d])))) \bowtie (Employee[e])
\end{aligned}
$$

The two transformations result from the variable binding convention and renaming, as well as the equivalence of d-join and join in case the inner argument of the d-join does not depend on the outer argument:

$$
\begin{aligned}
e_1 < e_2 > \quad &= \quad e_1 \bowtie e_2\\
&\quad\;\; \text{if } \mathcal{F}(e_2) \cap \mathcal{A}(e_1) = \emptyset
\end{aligned}
$$

Note that the join is in fact a cross product.

We proceed with our example by pushing $\chi_{ed:e.dept}$ inside and transforming the crossproduct into a join

$$
\begin{aligned}
e_2 &\equiv \pi_d(\sigma_{dl=\text{``}Paris\text{''}}(\chi_{dl:dd.locatedIn.city}(Department[d])))\\
e_3 &\equiv \chi_{ed:e.dept}(Employee[e])\\
q &\equiv \chi_e(\sigma_{d=ed}(e_2 \bowtie e_3))\\
&\equiv \chi_e(e_2 \bowtie_{d=ed} e_3)
\end{aligned}
$$

where the last join actually is a semi-join which can be efficiently processed.

Aggregated Nested Block Within the Where Clause

Aggregated nested blocks also occur in the relational context. We treat them in similar manner although we rely on set attributes to avoid dealing with null values. Let us consider the following query:

```
select  d
from    d in Department
where   count(select  e
              from    e in Employee
              where   e.dept == d)
        > 10
```

```
select  d
from    d in Department
where   c > 10
define  c = count(select  e
                  from    e in Employee
                  where   ed == d
                  define  ed = e.dept)
```

The rewriting is similar to the already discussed case of nesting in the where clause with no aggregates applied:

$$
\begin{aligned}
q &\equiv \chi_d(\sigma_{c>10}(\chi_{c:e_2}(Department[d])))\\
e_2 &\equiv count(\chi_e(\sigma_{d=ed}(\chi_{ed:e.dept}(Employee[e]))))\\
q &\equiv \chi_d(\sigma_{c>10}(Department[d] \mathbin{\sqsupset\!\!\bowtie}^{c=0}_{d=ed} (\Gamma_{c;ed;count\circ\chi_c}(\chi_{ed:e.dept}(Employee[e])))))
\end{aligned}
$$

In case the outerjoin is followed by a selection which evaluates to false for the default value, the outer join can be replaced by a join. Thus we have:

$$q \equiv \chi_d(\sigma_{c>10}(Department[d] \bowtie_{d=ed} (\Gamma_{c;ed;count\circ\chi_c}(\chi_{ed:e.dept}(Employee[e])))))$$

Furthermore, as in the first example, we may rely on semantics knowledge to skip the join operation. Indeed, if we know that all departments have at least one employee (i.e., we will access all departments from the Employee extent) and since all the information in the left argument of the equation is already present in the right argument (since $d == ed$) we may rewrite the query in the following way:

$$q \equiv \chi_{ed}(\sigma_{c>10}(\Gamma_{c;ed;count\circ\chi_c}(\chi_{ed:e.dept}(Employee[e]))))$$

Special Cases for Min and Max

There exist special cases where selections based on aggregates can be treated more efficiently. We will show that with the following query with two levels of nesting and a *max* function applied:

```
select tuple(name: e.name,
       bestSales: (select s
                   from   s in e.sales
                   where  s.amount() =
                          max(select s.amount()
                              from   s in e.sales)))
from   e in Employee
```

```
select tuple(name: en, bestSales: bs)
from   e in Employee
define en = e.name
       es = e.sales
       m = max(select tv
               from   t in es
               define tv = t.amount())
       bs = select s
            from   s in es
            where  sv = m
            define sv = s.amount()
```

Note that the dependency-based optimization already took one level of nesting off. The translation to the algebra yields:

$$\begin{aligned} q &\equiv \chi_{[name:en,bestSales:bs]}(\chi_{bs:e_2}(\chi_{m:e_1}(\chi_{en:e.name,es:e.sales}(Employee[e])))) \\ e_1 &\equiv max(\chi_{tv}(\chi_{tv:t.amount()}(es[t]))) \\ e_2 &\equiv \chi_s(\sigma_{sv=m}(\chi_{sv:s.amount()}(es[s]))) \end{aligned}$$

Here, we may apply the following equivalence:

$$\chi_{g:f(\sigma_{a=m}(e_1))}(\chi_{m:max(e_2)}(e)) = \chi_{\circ(max_{g;m;a;f}(e_1))}(e) \quad \text{if } e_1 = \pi_a(e_2)$$

We remind that the $max_{g;m;a;f}(e)$ operation returns a tuple containing (i) an attribute m representing the maximum value for the attribute a in the set e and (ii) an attribute g representing the result of f applied to the set of elements of e whose attribute a is equal to m. The above equivalence applied to the query yields:

$$q \equiv \chi_{[name:en,bestSales:bs]}(\chi_{\circ(max_{bs;m;sv;\chi_s}(\chi_{sv:s.amount()}(es[s])))}(\chi_{en:e.name,es:e.sales}(Employee[e])))$$

Note that the *max* function can be computed in a single scan (linear time) for $max_{g;m;a;f}$ if f is linear.

Also note that an equivalent treatment for *min* can be applied. Furthermore, although the equivalence we used can easily be adapted to the relational context, we are not aware of any such optimization.

6 Conclusion

In this paper, we showed that much could be done for the optimization of nested queries in the object-oriented context. We have proposed a two-phase optimization. The first phase transformed a nested query at the source level in order to treat common subexpressions and those not dependent on the current block more efficiently. After the translation of the transformed queries to the nested algebra, the second phase used algebraic equations in order to unnest nested algebraic expressions. This opened the road for more efficient evaluations.

We are now working on a classification of oo nested queries. The main goal behind this classification is to help the optimizer in choosing the appropriate equivalences. Furthermore, it will help to better see the analogies and differences between relational and oo nested queries, thus allowing each world to benefit from the other. Our current studies also concern multi-layered nested queries among which more efficient treatment of special cases of tree queries (in the spirit of [20]).

The topics for further research can be divided into two classes. The first class is concerned with implementation. To this class belong the extension of existing cost models and search strategies to the nested algebra. The second class contains topics involved in extending the current approach in order to incorporate bags and lists on the data model side as well as quantifiers on the query language side. Also, we consider using inverse functions for optimizing nested queries.

Acknowledgements: The authors thank Serge Abiteboul and Victor Vianu for many valuable comments on a first draft of the paper.

References

[1] M. M. Astrahan and D. D. Chamberlin. Implementation of a structured English query language. *Communications of the ACM*, 18(10):580–588, 1975.

[2] F. Bancilhon, S. Cluet, and C. Delobel. A query language for the o_2 object-oriented database system. In *DBPL II*, Salishan Lodge, Oregan, 1989.

[3] D. Beech. A foundation for evolution from relational to object databases. In *EDBT*, 1988.

[4] C. Beeri and Y. Kornatzky. Algebraic optimization of object-oriented query languages. In *Proc. Int. Conf. on Database Theory (ICDT)*, pages 72–88, 1990.

[5] M. Carey, D. DeWitt, and S. Vandenberg. A data model and query language for EXODUS. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 413–423, 1988.

[6] S. Ceri and G. Gottlob. Translating SQL into relational algebra: Optimization, semantics and equivalence of SQL queries. *IEEE Trans. on Software Eng.*, pages 324–345, 1985.

[7] S. Cluet and C. Delobel. A general framework for the optimization of object-oriented queries. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 383–392, 1992.

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

[9] O. Deux. The story of o2. *IEEE Transaction on Knowledge and Data Engineering*, 2(1), March 1989.

[10] G. Lohman et al. Optimization of nested queries in a distributed relational database. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, 1984.

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

[12] W. Hasan and H. Pirahesh. Query rewrite optimization in starburst. Research Report RJ6367, IBM, 1988.

[13] P. Jenq, D. Woelk, W. Kim, and W. Lee. Query processing in distributed ORION. In *Proc. Int. Conf. on Extended Database Technology (EDBT)*, Venice, 1990.

[14] A. Kemper and G. Moerkotte. Advanced query processing in object bases using access support relations. In *Proc. Int. Conf. on Very Large Data Bases*, pages 294–305, 1990.

[15] A. Kemper and G. Moerkotte. Query optimization in object bases: Exploiting relational techniques. In *Proc. Dagstuhl Workshop on Query Optimization (J.-C. Freytag, D. Maier und G. Vossen (eds.))*. Morgan-Kaufman, to appear 1993.

[16] A. Kemper, G. Moerkotte, and K. Peithner. A blackboard architecture for query optimization in object bases. Informatik-Fachberichte 92-31, RWTH Aachen, 5100 Aachen, Germany, 1992.

[17] W. Kiessling. SQL-like and Quel-like correlation queries with aggregates revisited. ERL/UCB Memo 84/75, University of Berkeley, 1984.

[18] W. Kim. A model of queries for object-oriented database. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, 1989.

[19] W. Kim. On optimizing an SQL-like nested query. *ACM Trans. on Database Systems*, 7(3):443–469, Sep 82.

[20] M. Muralikrishna. Optimization and dataflow algorithms for nested tree queries. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, 1989.

[21] M. Muralikrishna. Improved unnesting algorithms for join aggregate SQL queries. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 91–102, 1992.

[22] A. Rosenthal and C. Galindo-Legaria. Query graphs, implementing trees, and freely-reorderable outerjoins. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 291–299, 1990.

[23] G.M. Shaw and S.B. Zdonik. A query algebra for object-oriented databases. In *Proc. IEEE Conference on Data Engineering*, pages 154–162, 1990.

[24] D. Straube and T. Özsu. Queries and query processing in object-oriented database systems. *ACM Trans. on Information Systems*, 8(4):387–430, 1990.

[25] S. L. Vandenberg and D. DeWitt. Algebraic support for complex objects with arrays, identity, and inheritance. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 158–167, 1991.