Classification and Optimization of Nested Queries in Object Bases

Sophie Cluet *Guido Moerkotte*

INRIA
BP 105
Domaine de Voluceau
78153 Le Chesnay Cedex
France
Sophie.Cluet@inria.fr

Fakultät für Informatik
Universität Karlsruhe
Am Fasanengarten 5
D–7500 Karlsruhe
Germany
moer@ira.uka.de

February 16, 1994

Abstract

Many declarative query languages for object-oriented (oo) databases allow nested subqueries. This paper contains a complete classification of oo nested queries and appropriate unnesting optimization strategies based on algebraic rewriting. We adapt some known relational techniques and introduce new ones that use and are concerned with features specific to object-oriented queries. In particular, we introduce two new and powerful grouping operators which will form the basis for our unnesting techniques.

Keywords: Query optimization, Object Oriented Databases.

1 Introduction

One essential feature of declarative query languages is the nesting of queries (embedding of a query into another query). In SQL, nested queries are used to express complex conditions. This is also true in object-oriented (oo) SQL-like query languages. But oo nested queries serve other purposes as well: they are used to access nested structures and to produce nested results. Thus, they represent a fundamental characteristic of oo query languages. Yet, their optimization has remained largely unexplored. We propose novel techniques for efficiently evaluating nested queries in oo languages.

In the last few years, many algebras have been proposed for the optimization of oo query languages [2, 3, 7, 16, 28, 30, 31]. Most of them allow the representation of nested algebraic expressions (e.g., a join may occur within the predicate of a selection). However,

the proposed algebraic equivalences do not deal with these nested parts that remain nested throughout the rewriting process. Finally, they are evaluated by rather inefficient nested loops.

We propose a complete classification of oo nested queries together with appropriate algebraic unnesting techniques. It is fundamental to understand that unnesting a query is not to return an unnested structure, but an unnested algebraic expression (e.g., no join within the predicate of a selection). The interest of unnesting queries is twofold. Unnested expressions can be evaluated more efficiently and they can be rewritten using standard equivalences. Our unnesting technique heavily builds on two powerful grouping operators. The *unary grouping* is a generalization of the common NF^2 nest operator [27]. The *binary grouping* operator is even more powerful and is used to more elegantly unnest queries where an outerjoin followed by a unary grouping is needed. The binary grouping operator generalizes the *nest-join* operator of [29].

The optimization of relational nested queries has been studied thoroughly in the last decade [20, 19, 13, 10, 14, 24, 26, 25]. Naturally, our research has been influenced by this body of work. The classification of relational nested queries introduced in [20] proved to be useful for relational unnesting. Based on this observation, we extended this classification to the oo context. This extension is necessary due to the following observations. As opposed to SQL where the **where** clause is the only place for occurring nested queries, all clauses are equally important in an oo SQL-like language. That is: (i) nested blocks may be located in any clause of a query and (ii) a dependency (i.e., reference to a variable of the outer block) may occur in any clause of a query inner block. Our classification takes this orthogonality of oo SQL-like query languages into account.

Also, the relational idea of using different join operators for unnesting nested queries will be carried over to the oo context [20, 10, 14, 24]. However, as opposed to the relational context, where unnesting is performed at the SQL level, we use algebraic equivalences. Thus, our work can be applied to different query languages, if a translation procedure into the algebra is given. Further, we will show that, due to the nesting in the oo data structures ($\neg$1NF), there exist more elegant and efficient alternatives for unnesting queries. First, we can avoid null values. They are needed in the relational context to represent misfits in the joins (outer-joins) introduced by the unnesting process. This is more naturally captured by empty sets in the oo context. Secondly, the **group-by** of SQL is captured in the relational context in a distorted manner using operations that couple grouping and aggregate. In the oo context, explicit grouping operations are a necessity but they also yield more flexibility for expressing and rewriting queries. This results in alternative unnesting techniques not known in the relational context.

Due to space limitation, this paper concentrates on one level nested queries. More complex unnesting strategies including the treatment of several levels of nesting, non-neighbor predicates and outer restrictions can be found in [9]. The paper is organized as follows. The next section introduces some preliminaries, namely the algebra and the translation of SQL-like queries into the algebra. The core of the paper is contained in Section 3. It represents the classification of oo nested queries together with the algebraic equivalences used for unnesting. We review some results of [8] and introduce new unnesting techniques for those cases which could not be unnested so far. In Section 4, we briefly present techniques for unnesting nested queries with quantifiers. More on the treatment of quantifiers can be found in [9]. Section 5 concludes the paper.

2 Preliminaries

2.1 Algebra

We assume standard knowledge on oo data models. Our underlying data model is similar to the O_2[11], GOM [16] or Exodus [4] model and conforms to the Object Database Standard [5]. 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 extension which is a set containing the object identifiers of all its instances. The model also features complex values (or literals) with no identity, that are manipulated by standard operators and do not belong to classes. Hence, there are no extensions for them.

The algebra is an extension of the GOM algebra [17, 18]. Its main characteristic is that — with the exception of the map operator — it is defined on sets of tuples. This guaranties some nice properties among which is the associativity of the join operator.

We now restate the algebraic operators that will be used in the sequel. Union, intersection and difference operators, that are part of the algebra, are not reviewed.

Map Operations (and Projection) These operators are fundamental to the algebra. Since the other operators are defined on sets of tuples, sets of non-tuples (mostly sets of objects) must be transformed into sets of tuples. This is one purpose of the map operator. Other purposes are dereferenciation, method and function application. Also, our translation process pushes all nesting into map operators.

The first definition corresponds to the standard map [17] or materialize [3] operator. The second and third definition are just shorthands: the second is a map with tuple concatenation and the third a map with tuple construction.

$$\begin{aligned} \chi_{e_2}(e_1) &= \{e_2(x)|x \in e_1\} \\ \chi_{a:e_2}(e_1) &= \{y \circ [a : e_2(y)]|y \in e_1\} \\ e[a] &= \{[a : x]|x \in e\} \end{aligned}$$

In the definitions, the e_i's denote both expressions (in the left hand side) and their evaluation (in the right hand side). Note that the oo map operator obviates the need of a relational projection. Sometimes the map operator is equivalent to a simple projection or renaming. In these cases, we will use π instead of χ.

Selection Note that in the following definition there is no restriction on the selection predicate. It may contain path expressions, method calls, nested algebraic operators, etc.

$$\sigma_p(e) = \{x|x \in e, p(x)\}$$

Join Operations The algebra features five join operators. Besides the complex join predicate, the first four of them are rather standard: join, semi-join, anti-join and left outer-join are defined similarly to their relational counterparts. One difference is that the left outer-join accepts a default value to be given, instead of null, to one attribute of its right argument.

$$
\begin{aligned}
e_1 \bowtie_p e_2 &= \{y \circ x | y \in e_1, x \in e_2, p(y,x)\} \\
e_1 \triangleright\!\!<_p e_2 &= \{y | y \in e_1, \exists x \in e_2, (p(y,x))\} \\
e_1 \triangleleft\!\!\triangleright_p e_2 &= \{y | y \in e_1, \neg\exists x \in e_2\ p(y,x)\} \\
e_1 \supset\!\!\bowtie {}^{g=c}_{p} e_2 &= \{y \circ x | y \in e_1, x \in e_2, p(y,x)\} \cup \\
&\quad \{y \circ z | y \in e_1, \neg\exists x \in e_2\ p(y,x), \mathcal{A}(z) = \mathcal{A}(e_2), g \in \mathcal{A}(e_2), \\
&\quad z.g = c, \forall a \in \mathcal{A}(e_2)(a \neq g \Rightarrow z.a = NULL)\}
\end{aligned}
$$

The function $\mathcal{A}$ used in the last definition returns the set of attributes of a relation.

The last join operator is called *d-join* ($< \cdot >$). It is a join between two sets, where the evaluation of the second set may dependent on the first set. It is used to translate **from** clauses into the algebra. Here, the range definition of a variable may depend on the value of a formerly defined variable. Whenever possible, d-joins are rewritten into standard joins.

$$
e_1 < e_2 > \ = \ \{y \circ x | y \in e_1, x \in e_2(y)\}
$$

Grouping Operations Two grouping operators will be used for unnesting purposes. The first one — called *unary grouping* — is defined on a set and its subscript indicates (i) the attribute receiving the grouped elements (ii) the grouping criterion, and (iii) a function that will be applied to each group.

$$
\begin{aligned}
\Gamma_{g;A\theta;f}(e) &= \{y.A \circ [g : G] | y \in e, \\
&\qquad G = f(\{x | x \in e, x.A\theta y.A\})\}
\end{aligned}
$$

Note that the traditional nest operator [27] is a special case of unary grouping. It is equivalent to $\Gamma_{g;A=;id}$. Note also that the grouping criterion may be defined on several attributes. Then, A and θ represent sequences of attributes and comparators.

The second grouping operator — called *binary grouping* — is defined on two sets. The elements of the second set are grouped according to a criterion that depends on the elements of the first argument.

$$
e_1 \Gamma_{g;A_1\theta A_2;f} e_2 \ = \ \{y \circ [g : G] | y \in e_1, G = f(\{x | x \in e_2, y.A_1\theta x.A_2\})\}
$$

In the sequel, the following abbreviations will be used: $\Gamma_{g;A;f}$ for $\Gamma_{g;A=;f}$, $\Gamma_{g;A}$ for $\Gamma_{g;A;id}$.

New implementation techniques have to be developed for these grouping operators. Obviously, those used for nest operator can be used for simple grouping when θ stands for equality. For the other cases, implementations based on sorting seem promising. We also consider adapting algorithms for non-equi joins, e.g. those developed for the band-width join [12].

Max Operation The Max operator has a very specific use that will be explained in the sequel. Note that an analogous Min operator can be defined.

$$
Max_{g;m;a\theta;f}(e) \ = \ [m : max(\{x.a | x \in e\}), g : f(\{x | x \in e, x.a\theta m\})]
$$

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

2.2 Translating SQL-like Queries into the Algebra

We use the O_2SQL language [1] for the examples. However, since we unnest at the algebraic level, the techniques that we present are not restricted to O_2SQL but can be applied to other languages like GOMql [16] or Excess [4]. As a matter of fact, O_2SQL is a subset of the oo standard query language as defined in [5]. The schema on which the queries are defined represents a company. It can be easily inferred from the queries and, hence, we will not detail it.

In [8], we showed the importance of factoring out constant or locally constant subqueries as well as common subexpressions. For this purpose, we introduced a phase preceding the optimization process. This phase — called *dependency-based optimization* — transforms the SQL-like query by introducing one variable per subquery. Thereby, common subexpressions are factorized; two occurrences of the same subquery are represented by only one variable. Further, expressions not dependent on the current block are pushed into higher level blocks.

Below is an O_2SQL query before (left hand-side) and after (right hand-side) the dependency-based transformation. To each employee's age, the query associates the number of sales made by younger employees.

```
select  tuple(age:x1.age,                          select  tuple(age:x1a, nb:cs)
              nb:count(select  s                   from    x1 in Employee
                       from    x2 in Employee      define  x1a = x1.age,
                               s in x2.sales               cs =  count(select  s
                       where   x2.age < x1.age))                           from    x2 in Employee
from    x1 in Employee                                                     s in x2.sales
                                                                           where   x2a < x1a
                                                                           define  x2a = x2.age
```

The transformed query features a new **define** clause in every block. Here, variables are introduced that represent expressions dependent on its owner block only. Thus, the operation $x2.age$ is now part of the outer block. For $x2.sales$. we did not introduce a variable. The motivation here is just to ease the translation process whose complete description is beyond the scope of this paper.

The following algebraic expression corresponds to the outer block of the query:

$$q \equiv \chi_{[age:x1a,nb:cs]}(\chi_{cs:e_2}(\chi_{x1a:x1.age}(Employee[x1])))$$

The **from** clause contains only one variable whose definition is translated by $Employee[x1]$. This allows us to view the set of object identifiers, i.e. the extension of *Employee*, as a unary relation with attribute $x1$. Note that this attribute allow us to avoid λ-expressions found in other oo algebras. The next two map (χ) operators correspond to the **define** clause. Their effect is to augment the initial relation with two more attributes representing the *age* and the result of the nested block (e_2). The last map operator corresponds to the **select** clause.

Let us now take a look at the inner block's algebraic expression.

$$e_2 \equiv count(\chi_s(\sigma_{x2a<x1a}(\chi_{x2a:x2.age}(Employee[x2] < x2.sales[s] >))))$$

The **from** clause of the inner block contains two variable definitions: this is translated by a d-join ($Employee[x2] < x2.sales[s] >$). Note that a standard join could not be used

since the evaluation of *x2.sales* obviously depends on the value of *x2*. The inner block features a **where** clause. This is represented — as might be expected — by a selection. Note that in this example, the last map operator (χ_s) is used to transform a relation into a set of objects.

In the sequel, for readability, we will sometimes combine map operations with other operators. We did that already in the previous example. We replaced the subexpression $(\chi_{x2s:x2.sales}(Employee[x2])) < x2s[s] >$ (that would have been derived if we had introduced a variable for $x2.sales$) by $Employee[x2] < x2.sales[s] >$ where map and d-join operations are combined.

3 Classification and Algebraic Optimization

In this section we present a classification of oo nested queries along with appropriate algebraic optimization techniques. The classification extends the relational classification of [20] which introduced five types of nested queries. One of them is not used here (Type D). The reason is that unnesting of Type D queries is subsumed by our generalized treatment of Type N and J queries. The four remaining types are:

- **Type A** nested queries have a constant inner block that returns a single element.
- **Type N** nested queries have a constant inner block that returns a set.
- **Type J** nested queries have an inner block that is dependent on the outer block and returns a set.
- **Type JA** nested queries have an inner block that is dependent on the outer block and returns a single element.

Obviously, the need for extending the original classification arises from the richness of the oo model compared to the relational one and its impact on the query language. The classification we propose has three dimensions: the original one plus two that are required by the following oo characteristics. In the oo context, as opposed to the relational, (i) nested blocks may be located in any clause of a **select-from-where** block and (ii) a dependency (i.e., reference to a variable defined in an outer block) may occur in any clause. The organization of this section follows the three dimensions. We start by presenting nested queries of Type A/N/J/JA with nesting and dependency in the **where** clause. This will allow us to show the differences and similarities between relational and oo unnesting techniques. We continue by explaining the treatment required by other locations of nesting (i.e., **select** and **from** clauses). We end with the optimization of Type J/JA queries with range (**from** clause) or projection (**select** clause) dependencies.

3.1 Different Types of Nesting

3.1.1 Queries of Type A

Below is an example of a type A query. It returns the employees with the maximum sales. The inner block is constant and returns a single numerical value.

```
select  x1                                  define m = max(select  x2s
from    x1 in Employee                                     from    x2 in Employee)
where x1.TotSales =                                        define x2s = x2.TotSales
        max (select  x2.TotSales            select  x1
             from    x2 in Employee)        from    x1 in Employee
                                            where x1s = m
                                            define x1s = x1.TotSales
```

Note that the first **define** clause is written before the outer block because its entry does not depend on any variable. The algebraic translation of the query is:

$$\begin{aligned} q &\equiv \chi_{x1}(\sigma_{x1s=m}(\chi_{x1s:x1.TotSales}(Employee[x1]))) \\ m &\equiv max(\chi_{x2s}(\chi_{x2s:x2.TotSales}(Employee[x2]))) \\ &\equiv max(\chi_{x2.TotSales}(Employee[x2])) \end{aligned}$$

For most queries of this kind, we rely on a technique similar to that of [20]. The constant inner block (m in the example) is evaluated first and its result is used for the evaluation of the outer block (q in the example).

However, for this query, as well as all min/max queries where inner and outer blocks have a common domain ($Employee$ in the example), it is possible to do better than just pushing out the constant block [8]. The requirement of having a common domain seems rather restrictive but retrieving the elements exhibiting the min/max value implies just this. The idea is to use the scan needed for the evaluation of the inner block to also evaluate the outer block. This is exactly what the *Max* operator does. It is introduced by the following algebraic equivalence which is a slightly generalized version of the one found in [8]:

$$f(\sigma_{a\theta max(e_2)}(e_1)) \equiv Max_{g;m;a\theta;f}(e_1).g \quad \text{if } e_2 = \pi_a(e_1) \tag{1}$$

Remember that the $Max_{g;m;a\theta;f}(e)$ 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 satisfying $a\theta m$.

Applying Eqn. 1 to the above query yields

$$q \equiv Max_{g;m;x1s;\chi_{x1}}(\chi_{x1s:x1.TotSales}(Employee[x1])).g$$

Note that the *Max* operator can be computed within a single pass (linear time) for $Max_{g;m\theta;f}$, 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.

> In the general case, type A queries are treated, as in the relational context, by constant factorization. Special cases concern min/max operations that may have a more efficient evaluation by applying the *Min/Max* operators.

3.1.2 Queries of Type N

Below is an example of a Type N query. It returns the employees who have sold all the expensive products. As can be seen, the inner block is constant and, as opposed to type A queries, returns a collection that has to be scanned for every element of the outer block.

This inner scan is the reason for considering techniques different from that of type A queries.

```
select x                                    define ExpItems = select i
from   x in Employee                                          from   i in Item
where x.SoldItems ⊇ select i                                  where  p > 20000
                    from  i in Item                           define p = i.price
                    where i.price > 20000   select x
                                            from   x in Employee
                                            where xsi ⊇ ExpItems
                                            define xsi = x.SoldItems
```

Its algebraic translation is the following:

$$\begin{aligned} q &\equiv \chi_x(\sigma_{xsi \supseteq ExpItems}(\chi_{xsi:x.SoldItems}(Employee[x]))) \\ ExpItems &\equiv \chi_i(\sigma_{p>20000}(\chi_{p:i.price}(Item[i]))) \end{aligned}$$

In the above example, the predicate that connects outer and inner blocks is based on a set comparison. Set comparators are not supported by standard SQL. However, they are in the non-standard SQL of [20], where this query would be considered as **Type D** — resolved by division. Relational division implements very specific queries: i.e., there is a non strict inclusion relationship between outer and inner blocks (as in the above query). However, the probability to have a non-strict inclusion in a query is not higher than that of any other set comparator (or its negation) and, should we choose the division solution, we would have to introduce one division-like operation per set comparator. Furthermore, the division operation do not have particularly nice algebraic properties that we would like to exploit. This is why we rejected the Type D treatment applying division.

We chose to leave type N queries with set comparisons as they are. We will explain our choice using the above query. First, note that the nesting itself has been solved by constant factorization. The nested query $ExpItems$ can be evaluated first, independently of expression q. Secondly, remember that, in the oo context, attributes may be sets. Thus, comparing set attributes in algebraic operations will arise commonly. Accordingly, it seems natural to support good algorithms for their evaluation.

Other possible predicates connecting inner and outer blocks are based on membership. For these, we could also rely on good implementations of the selection operator. We prefer to adapt techniques from the relational context [6, 10, 20] and use semi-join and anti-join for unnesting the **in** and **not in** cases. The interest for unnesting using join operations, apart from existing efficient algorithms, is that they have algebraic properties that can be used for further rewriting. Since they are not totally new, we just briefly cast these techniques into algebraic equivalences.

$$\sigma_{A_1 \in \chi_{A_2}(e_2)} e_1 \equiv e_1 \rhd\!\lhd_{A_1=A_2} e_2 \quad \text{if } A_i \subseteq \mathcal{A}(e_i),\ \mathcal{F}(e_2) \cap \mathcal{A}(e_1) = \emptyset \tag{2}$$

$$\sigma_{A_1 \notin \chi_{A_2}(e_2)} e_1 \equiv e_1 \blacktriangleleft\!\blacktriangleright_{A_1=A_2} e_2 \quad \text{if } A_i \subseteq \mathcal{A}(e_i),\ \mathcal{F}(e_2) \cap \mathcal{A}(e_1) = \emptyset \tag{3}$$

The first condition in the equations states that attribute A_1 (resp. A_2) must belong to expression e_1 (resp. e_2). The $\mathcal{F}$ function returns the free variables of an expression. Thus,

the condition "$\mathcal{F}(e_2) \cap \mathcal{A}(e_1) = \emptyset$" states that the inner expression e_2 must not depend on the outer expression e_1 (as is always the case in type N queries).

> Type N queries are unnested by constant factorization. Queries with **in** and **not in** predicates can be further rewritten, as in the relational context, using semi-joins and anti-joins. For queries with set comparators we rely on good implementations of the selection operation with set predicates (needed anyway since attributes may be sets in the oo context).

3.1.3 Queries of Type J

In Type J, as opposed to Type N, the inner query is dependent on a variable of the outer block. Thus, it cannot be factored out. An example is given below. As for type N queries, the predicate connecting inner and outer blocks is based either on membership or on a set comparator. Again, queries with **in** (**not in**) can be transformed using semi-joins (anti-joins). Some differences occur due to the dependency between the two blocks (see the second condition below). Type J **not in** queries cannot be translated directly using an anti-join operation: a semi-join has to be performed first.

$$\sigma_{A_1 \in \chi_{A_2}(\sigma_p(e_2))} e_1 \equiv e_1 \ltimes_{A_1=A_2 \wedge p} e_2 \quad (4)$$
$$\text{if } A_i \subseteq \mathcal{A}(e_i),\ \mathcal{F}(p) \subseteq \mathcal{A}(e_1 \cup e_2),\ \mathcal{F}(e_2) \cap \mathcal{A}(e_1) = \emptyset$$

$$\sigma_{A_1 \notin \chi_{A_2}(\sigma_p(e_2))} e_1 \equiv e_1 \,\overline{\ltimes}_{A_1=A_2}\, (e_2 \ltimes_p e_1) \quad (5)$$
$$\text{if } A_i \subseteq \mathcal{A}(e_i),\ \mathcal{F}(p) \subseteq \mathcal{A}(e_1 \cup e_2),\ \mathcal{F}(e_2) \cap \mathcal{A}(e_1) = \emptyset$$

Now let us consider queries with set comparison. These also correspond to relational Type D queries. The query below returns the employees who have done all the expensive sales of their department.

```
select x                                  select x
from   x in Employee                      from   x in Employee
where  x.sales ⊇ select s                 where  xs ⊇ BestSales
                 from   s in Sale         define xs = x.sales
                 where  s.dept=x.dept and          xd = x.department
                        s.amount>2000              BestSales = select  s
                                                               from    s in Sale
                                                               where   sd=xd and sa > 2000
                                                               define  sd=s.dept
                                                                       sa=s.amount
```

The algebraic translation of the query is split for clarity:

$$\begin{aligned} q &\equiv \chi_x(\sigma_{xs \supseteq BestSales}(q_1)) \\ q_1 &\equiv \chi_{BestSales:\chi_s(\sigma_{sd=xd}(q_3))}(q_2) \\ q_2 &\equiv \chi_{xd:x.dept,xs:x.sales}(Employee[x]) \\ q_3 &\equiv \sigma_{sa>2000}(\chi_{sd:s.dept,sa:s.amount}(Sale[s])) \end{aligned}$$

The problem here is that the nested query evaluating $BestSales$ is not constant. In order to unnest the query and avoid several costly scans over the *Sale*s extension, we associate

with each employee, the set of expensive sales of his/her department. For this, we rely on the following equivalence. It uses binary grouping.

$$\chi_{g:f(\sigma_{A_1\theta A_2}(e_2))}(e_1) \equiv e_1\Gamma_{g;A_1\theta A_2;f}e_2 \quad \text{if } A_i \subseteq \mathcal{A}(e_i),\ \mathcal{F}(e_2)\cap\mathcal{A}(e_1)=\emptyset,\ g\notin A_1\cup A_2 \tag{6}$$

The conditions state that the A_i's are attributes of the e_i's, e_1 and e_2 must be independent, and g must not be an attribute of e_1 or e_2. Applying Eqn. 6 to q_1 yields:

$$q_1 \equiv q_2\Gamma_{BestSales;xd=sd;\chi_s}q_3$$

If employees and sales are both sorted on their department, this binary group operation can be evaluated very efficiently in a single scan over the two sets. This simple evaluation is not possible when the grouping criterion is based on an operation different than equality. The selection in q_3 can be evaluated independently, e.g. using an index, or can be combined with the χ_s operator in the group. Note that the selection with set comparator $\supseteq$ is now evaluated between two attributes. As for type N queries, we rely on good algorithms for such selections.

The above equation is the most general and can be applied in all Type J queries (as well as to Type JA cases as will be shown in the sequel). There exist two other equations which may deal more efficiently, using simple grouping, with two special cases. The equation

$$\chi_{g:f(\sigma_{A_1=A_2}(e_2))}(e_1) \equiv \pi_{\overline{A_2}}(e_1 \overset{g=f(\emptyset)}{\supset\!\!\bowtie}_{A_1=A_2}(\Gamma_{g;A_2;f}(e_2))) \quad \text{if } A_i \subseteq \mathcal{A}(e_i),\ \mathcal{F}(e_2)\cap\mathcal{A}(e_1)=\emptyset,\ A_1\cap A_2=\emptyset, g\notin\mathcal{A}(e_1)\cup\mathcal{A}(e_2) \tag{7}$$

relies on the fact that the comparison of the correlation predicate is equality, as in the above example. The expression $\pi_{\overline{A_2}}$ eliminates the attribute(s) A_2 from the relation it is applied to. If we apply this equation on the original q_1 expression, it yields a new alternative:

$$q_1 \equiv \pi_{\overline{sd}}(q_2 \overset{BestSales=\emptyset}{\supset\!\!\bowtie}_{sd=xd}(\Gamma_{BestSales;sd;\chi_s}(q_3)))$$

Note that the binary grouping operation in the previous expression of q_1 now corresponds to a simple grouping followed by an outer-join (see below, also). The outer-join does not introduce null values for those employees whose department do not have expensive sales. The $\emptyset$ default value corresponds to the application of the operation χ_s on the empty set. The $\pi_{\overline{sd}}$ expression can be dropped due to the χ_x operation in q.

The next equation relies on the fact that there exists a common range for the variables of the correlation predicate (third condition). Its application will be illustrated in a subsequent subsection.

$$\chi_{g:f(\sigma_{A_1\theta A_2}(e_2))}(e_1) \equiv \pi_{A_1:A_2}(\Gamma_{g;A_2\theta;f}(e_2)) \quad \text{if } A_i \subseteq \mathcal{A}(e_i),\ \mathcal{F}(e_2)\cap\mathcal{A}(e_1)=\emptyset,\ g\notin\mathcal{A}(e_1)\cup\mathcal{A}(e_2),\ e_1=\pi_{A_1:A_2}(e_2)\ \text{(this implies that } A_1=\mathcal{A}(e_1)) \tag{8}$$

It is important to note that the applicability of the above equivalences does not depend on the set comparison of the predicate in the outer **where** clause but, instead, on the

correlation predicate within the inner block. This is opposed to the relational unnesting techniques which solely depend on the predicate restricting the outer block. This has one important consequence: a uniform treatment, independent of the predicate in the outer block, is possible. Hence, they enable the derivation of alternative unnested expressions for the **in** and **not in** cases. To see this, consider $\sigma_{A \in e_2}(e_1) \equiv \sigma_{A \in B}(\chi_{B:e_2}(e_1))$. Further, as illustrated in the next subsection, the equations can be used (unchanged!) for unnesting Type JA nested queries. That is why they should be considered *the core* of unnesting nested queries in the oo context. Also, since these equations also hold for set comparators as the predicate in the outer block, they obviate a special treatment of Type D queries as needed in the relational case.

Let us finally clarify the correspondence between the binary grouping and the unary grouping in conjunction with an outer-join:

$$\chi_{g:f(\sigma_{A_1' \theta A_2'}(e_2))}(e_1) \equiv \Gamma_{g;A_1;f \circ \pi_{\overline{A_1}} \circ \sigma_{A_2 \neq \perp_{A_2}}}(e_1 \overset{\supset}{\bowtie}_{A_1' \theta A_2'} e_2) \quad (9)$$
$$\text{if } A_i = \mathcal{A}(e_i),\ A_i' \subseteq A_i,\ g \notin A_1 \cup A_2,\ \mathcal{F}(e_2) \cap A_1 = \emptyset$$

Here, $\perp_A$ is a tuple with attributes A and null values only. This alternative first produces an intermediate flat result which is subsequently grouped. The range of applicability of this equation is the same as for Eqn. 6 which introduces a binary grouping. Concatenating Eqns. 6 and 9 gives the correspondence between binary and unary grouping in conjunction with an outer-join. If several of the Eqns. 6–9 apply, chosing the most appropriate is a matter of costs.

> In the general case, Type J queries are unnested using the binary grouping operator or, alternatively, an outer-join followed by a unary grouping. Other alternatives are possible for specific cases. Queries with membership predicates can be evaluated using semi-joins and anti-joins. If its inner and outer blocks have a common domain, a nested query can be rewritten into a simple grouping. A query with a correlation predicate on equality can be rewritten into simple grouping followed by an outer-join.

3.1.4 Queries of Type JA

The difference between Type J and Type JA queries it that, for the latter, an aggregate function (e.g, count, max) is applied on the inner block.

However, this difference between Type J and Type JA queries does not necessitate new unnesting techniques. Indeed, the grouping operators, that we used for unnesting type J queries, have been defined to allow the application of an arbitrary function to each formed group. For Type J queries, this function usually is a map operator. For Type JA queries this is an (additional) aggregate function. Thus, by applying Equations 6–9, aggregated Type JA queries are treated in exactly the same manner as Type J queries. The following subsection will give an example.

It is interesting to note that Eqn. 7, when used on Type JA queries (i.e., with an aggregate function), captures the technique introduced by [10] for relational type JA queries. His technique relies on an operator called *generalized aggregation* combining a grouping with an aggregate.

In the oo context, using the full power of the two grouping operators, Type JA queries are unnested in the same manner as Type J queries.

3.2 Different Locations of Nesting

So far, we studied queries where the nesting was used to express complex conditions (nesting in the **where** clause). Nesting has other purposes in the oo context, too. It can be used to produce nested results (**select** clause) or access nested structures (**from** clause). This is what we study now.

3.2.1 Nesting in the *select* Clause

Although nothing forbids it, Type A or N nesting rarely occurs in **select** clauses. Indeed, there is not much sense in associating a constant (set or element) to each element of a set. Should that happen, we rely on the first phase of the optimization process to factor out the constant block. Thus, it will only be evaluated once. We do not believe in or consider further optimization.

For Type J/JA queries, nesting in the **select** clause is equivalent to nesting in the **where** clause. Remember that the application of Eqns. 6–9 did not depend on the predicate in the outer **where** block but on the correlation predicate within the inner block. I.e., it did not depend on the location of the nesting, but on the fact that a dependency between inner and outer block existed in the form of a selection. Hence, Eqns. 6–8 can be used whenever nesting occurs in the **select** clause. To illustrate this, we take the query of the previous Section (Page 5). It is of Type JA and associates to each employee's age, the number of sales made by younger employees. The algebraic expression is the following:

$$\begin{aligned} q &\equiv \chi_{[age:x1a,nb:cs]}(\chi_{cs:e_2}(\chi_{x1a:x1.age}(Employee[x1]))) \\ e_2 &\equiv count(\chi_s(\sigma_{x2a<x1a}(\chi_{x2a:x2.age}(Employee[x2] < x2.sales[s] >)))) \end{aligned}$$

Assume that every employee has at least one sale. Then, we may use advantageously the fact that one variable (x2) of the inner block ranges over the same set (all the employees) than the variable of the outer block (x1) to apply equivalence 8.

$$\begin{aligned} q &\equiv \chi_{[age:x1a,nb:cs]}(\pi_{x1a:x2a}\Gamma_{cs;x2a\leq;count\circ\chi_s}(e_2')) \\ e_2' &\equiv (\chi_{x2a:x2.age}Employee[x2]) < x2.sales[s] > \end{aligned}$$

Note that we pushed the map operation before the d-join in expression e_2'. 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.

There exists one Type J case where another more powerful technique can be applied: a *flatten* operation is performed on the outer block, and there is no tuple constructor within the outer block's **select** clause. As shown in [8], these queries can be optimized by pushing the *flatten* operation inside until it is applied on stored attributes, thus eliminating the nesting.

Nesting in the **select** clause can be optimized as nesting in the **where** clause. If a **flatten** operation is used on the outer-block we can sometimes do better by pushing the flatten operation inside.

3.2.2 Nesting in the *from* Clause

For the moment, we consider that inner blocks are constant or that their dependency on outer blocks are expressed through a selection. Other dependencies are studied in the sequel. In the current context, nesting in the **from** clause is easily optimized. Due to space limitation, we do not illustrate this with an example but the following two equivalences should clarify the matter.

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

Let us consider that $e_1$and e_2 are, respectively, the outer and inner block of an O_2SQL query. Since the nesting is located in the **from** clause, this results in a d-join between e_1 and e_2. If there is a predicate dependency between inner and outer block, it can be pushed out of the d-join (first equivalence). When no dependency exists between inner and outer blocks, the d-join can be transformed into a standard join (second equivalence).

Nesting in the **from** clause can be optimized by transforming d-join operations into standard joins.

3.3 Different Kinds of Dependency

Remember that we distinguish three kinds of dependencies: projection dependency (a reference to an outer variables occurs in the **select** clause), range dependency (...in the **from** clause) and predicate dependency (...in the **where** clause). Above, we studied queries with predicate dependency. In the sequel, we concentrate on optimization techniques required for range and projection dependencies.

3.3.1 Range Dependency

Consider the following query exhibiting a range dependency. It associates to each employee, the name of their customers living in Roma. The inner block's variable s depends on the outer block.

```
select tuple (e: x.name, c: select s.customer.name
                            from   s in x.sales
                            where  s.customer.city
                                     = "Roma")
from   x in   Employee
```

```
select   tuple (e: xn, c: SCN)
from     x in Employee,
define   xn = x.name
         xs = x.sales
         SCN = select   scn
               from     s in xs
               where    scc = "Roma"
               define   sc = s.customer
                        scn = sc.name
                        scc = sc.city
```

Translation into the algebra yields:

$$\begin{aligned} q &\equiv \chi_{[e:xn,c:SCN]}(\chi_{SCN:nq}(\chi_{xn:x.name,xs:x.sales}(Employee[x]))) \\ nq &\equiv \chi_{scn}(\sigma_{scc="Roma"}(\chi scc : sc.city(\chi_{scn:sc.name}(\chi_{sc:s.customer}(xs[s]))))) \\ &\equiv \chi_{sc.name}(\sigma_{scc="Roma"}(\chi_{sc:s.customer,scc:sc.city}(xs[s]))) \end{aligned}$$

The only unnesting technique that can be applied to this query, and those of same kind, consists in reducing the range dependency into a predicate dependency. Once this is done, we can apply the above equations. The reduction, from range to predicate dependency, is based on *type based rewriting* as found in [7, 15, 21, 22, 23]. This technique relies on the existence of type extents for rewriting queries. Since type based rewriting itself has already been described, we just comment on its application to unnesting using the above example.

Relying on the fact that the elements of the attribute *sales* of an employee belong to the extent of the class *Sale*, the inner block of the query can be rewritten as

$$nq \equiv \chi_{sc.name}(\sigma_{scc=\text{“}Roma\text{”}}(\chi_{sc:s.customer,scc:sc.city}(\sigma_{s\in xs}(Sale[s]))))$$

Type based rewriting can be performed again using the extent of class Customer. This allows, for instance, to use indexes on *Customer.city* and *Sale.customer* to evaluate the query. However, since our goal is unnesting and not general optimization, we do not detail on this. Concerning unnesting, it is important to note that the dependency no longer specifies the range ($xs[s]$) but now represents a predicate ($\sigma_{s\in xs}$). Herewith, the algebraic expression is of the same form as one resulting from a predicate dependency. Hence, the above mentioned unnesting techniques apply.

However, in our model, the domain of an attribute is not always covered by an extent. Thus, we cannot always apply type based rewriting. In these cases, we do not unnest and rely on existing optimization techniques for path expressions crossing sets (e.g., x.sales.customer.city).

Range dependency can be transformed into predicate dependency using rewriting techniques based on extents.

3.3.2 Projection Dependency

In the algebra, the inner block of a query with projection dependency ends with a map operation with a function argument depending on both, the inner and the outer blocks. This kind of query should be rare which is good because they cannot be unnested. Nested loops or cross products have to be used for their evaluation. The only optimization that can be done is to push those parts of the expression that depends only on the outer block out of the inner block and to memorize (e.g., using an hash table) the results of the complex expressions depending solely on the inner block to avoid evaluating them only once.

Queries with projection dependency cannot be unnested.

Remarks Nothing restricts variables of the outer block to occur at only one place within the inner block. If there exist several dependencies, all the corresponding unnesting techniques can be applied alternatively. Hence, if, for example, a range and a predicate dependency occur, the latter should be used for unnesting if the range dependency cannot be resolved by type based rewriting.

Queries containing several nested blocks of same level are unnested by successive rewritings as in the relational context. Note that if the nesting occurs in a disjunctive **where** clause, the query has to be transformed first into a union.

Queries containing several levels of nesting are more complex. In the simple case, we can start the rewriting by the lower level and go up. However, when queries contain non-neighbor predicates, it is not always possible to apply the equivalences we introduced in the previous section.

Last not least, a query may contain quantifiers. This problem is treated briefly in the next section. Further techniques as well as a solution to non-neighbor predicates and an efficient treatment of outer restrictions can be found in a technical report [9].

4 Nested Queries with Quantifiers

In the previous section, we considered nested select-from-where (SFW) blocks. These blocks are not the only primitives allowing iterations over sets. Quantifiers also play a major role. This is why we have to consider nesting with quantifiers. We introduce two operators to deal with these. Due to space limitation, we only sketch these operators and appropriate unnesting strategies. More can be found in [9].

The operators are $\forall_p(e)$ and $\exists_p(e)$. They return *true* if all elements in e satisfy p or at least one element in e satisfies p, resp; else they return *false*.

Nested queries with quantifiers come in three different flavors. (i) The quantifier is nested in another block, (ii) a nested block is part of the quantifier condition, and (iii) a nested block is part of the domain of the quantifier. Let us consider the following example that corresponds to the first case. The query returns those employees having the same name as some other employee.

```
select  x1
from    x1 in Employee
where   exists x2
        in     Employee
        where  x2.name = x1.name and x1<>x2)
```

```
select  x1
from    x1 in Employee
where   EE
define  x1n = x1.name
        EE = exists x2
             in     Employee
             where  x1n=x2n and x1<>x2
             define x2n = x2.name
```

Translation to the algebra yields

$$\begin{aligned} q &\equiv \chi_{x1}(\sigma_{EE}(\chi_{EE:nq}(\chi_{x1n:x1.name}(Employee[x1])))) \\ nq &\equiv \exists_{x1n=x2n \wedge x1 \neq x2}(\chi_{x2n:x2.name}(Employee[x2])) \end{aligned}$$

Queries of this kind are treated in a manner similar to that of type J/JA queries. We simply have to adapt Eqns. 6–8 so that a quantifier replaces the selection used for the unnesting. In the above query, inner and outer block share a common domain. Thus we can adapt equation 7. The equivalence is given below. It is specified for existential quantifiers but an analogous equation can be defined for universal quantifiers.

$$\begin{aligned} \chi_{g:\exists_{A_1 \theta A_2}(e_2)}(e_1) \equiv\ & \pi_{A_1:A_2}(\Gamma_{g;\theta A_2;\exists_{true}(e_2)}) \\ & \text{if } A_i \subseteq \mathcal{A}(e_i),\ g \notin \mathcal{A}(e_1) \cup \mathcal{A}(e_2),\ \mathcal{F}(e_2) \cap \mathcal{A}(e_1) = \emptyset, \\ & e_1 = \pi_{A_1:A_2}(e_2) \text{ (implies } A_1 = \mathcal{A}(e_1)), \end{aligned}$$

Application to the query yields

$$\begin{aligned} q &\equiv \chi_{x1}(\sigma_{EE}(\pi_{x1n:x2n,x1:x2}(\Gamma_{EE;(=x2n,\neq x2);\exists_{true}}(\chi_{x2n:x2.name}(Employee[x2]))))) \\ &\equiv \chi_{x2}(\sigma_{EE}(\Gamma_{EE;(=x2n,\neq x2);\exists_{true}}(\chi_{x2n:x2.name}(Employee[x2])))) \end{aligned}$$

Note that, in the example, the group operation has two criterions. It associates a boolean value to each employee. The boolean value is "true" if the set of other employees ($\neq x2$) having same name ($= x2n$) is not empty ($\exists_{true}$). This query can be efficiently evaluated by, for instance, using an index on the employees name or by sorting the employees on their name and identifier.

Let us now consider, briefly, the cases where the quantifier is used as an outer block. These cases rarely occur by themselves but have to be considered for the treatment of multi-level nested queries. As stated above, the nesting can then be used for defining the quantifier predicate or the quantifier domain.

When the nested block is part of the condition of a quantifier, a treatment similar to that of SFW blocks with nesting in the **where** clause can be applied. This should be obvious for type A/N queries. Concerning type J/JA queries, remember that applying Equivalences 6, 7, or 8 did not rely on the predicate of the outer **where** block (in this case a quantifier) but on the correlation predicate within the inner block. Another technique can also be used for special cases. It consists in transforming the quantifier into a set comparison, using equivalences of the type

$$\forall_{\lambda x(x \in e_2)}(e_1) \equiv e_1 \subseteq e_2$$

When the nested block is used for defining the quantifier domain, there is no real unnesting to be done as is the case for nested blocks within a **from** clause. A more complete description on the treatment of quantifiers can be found in [9].

5 Conclusion

As opposed to the relational context where unnesting is performed at the SQL-level, we have introduced a technique which allows unnesting at the algebraic level. This frees us from query language dependencies. We introduced an algebra and showed how nested SQL-like queries are translated into nested algebraic expressions. Then, we reported several unnesting strategies in the form of algebraic equations. While some of them merely recast known relational strategies, most of them make use of and are concerned with features specific to object-oriented queries. Our presentation was concerned with one-level nested queries. More complex unnesting strategies including the treatment of several levels of nesting, non-neighbor predicates and outer restrictions can be found in [9].

There remain two topics for further research. The first topic concerns the implementation. Good algorithms for the extended operators have to be invented and existing cost models have to be extended to include these. A good starting point might be sorting based grouping operations. Special techniques for implementing non-equi joins (like the band-width join [12]) might also give good clues.

The second topic involves the extension of the current approach in order to incorporate bags and lists. As already indicated, a simple strategy is coding of these data structures with sets.

References

[1] 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.

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

[3] J. A. Blakeley, W. J. McKenna, and G. Graefe. Experiences building the oodb query optimizer. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 287–296, 1993.

[4] 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.

[5] R. Cattell, editor. *The Object Database Standard: ODMG 93.* Morgan Kaufmann, to appear.

[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] S. Cluet and G. Moerkotte. Nested queries in object bases. In *Proc. Int. Workshop on Database Programming Languages*, pages 226–242, 1993.

[9] S. Cluet and G. Moerkotte. Classification and optimization of nested queries in object bases. Technical report, University of Karlsruhe, 1994.

[10] 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.

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

[12] D. DeWitt, J. Naughton, and D. Schneider. An evaluation of non-equijoin algorithms. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, page 443, Barcelona, Spain, 1991.

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

[14] 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.

[15] 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.

[16] 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.

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

[18] A. Kemper, G. Moerkotte, and K. Peithner. A blackboard architecture for query optimization in object bases. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 543–554, 1993.

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

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

[21] G. Mitchell. *Extensible Query Processing in an Object-Oriented Database*. PhD thesis, Brown University, Providence, RI 02912, 1993.

[22] G. Mitchell, S. Zdonik, and U. Dayal. Object-oriented query optimization: What's the problem? Technical Report CS-91-41, Brown University, 1991.

[23] G. Mitchell, S. Zdonik, and U. Dayal. *Object-Oriented Database Systems (A. Dogac, M. T. Özsu, A. Biliris, and T. Sellis (eds.)-*, chapter Optimization of Object-Oriented Queries: Problems and Applicatios. NATO ASI. Springer, 1993. to appear.

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

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

[26] 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.

[27] H.-J. Schek and M. H. Scholl. The relational model with relation-valued attributes. *Information Systems*, 11(2):137–147, 1986.

[28] 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.

[29] H. J. Steenhagen, P. M. G. Apers, and H. M. Blanken. Optimization of nested queries in a complex object model. In *EDBT*, 1994. To appear.

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

[31] 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.