A General Framework for the Optimization of Object-Oriented Queries*

Sophie Cluet† and Claude Delobel †‡

Abstract

The goal of this work is to integrate in a general framework the different query optimization techniques that have been proposed in the object-oriented context. As a first step, we focus essentially on the logical aspect of query optimization. In this paper, we propose a formalism (i) that unifies different rewriting formalisms, (ii) that allows easy and exhaustive factorization of duplicated subqueries, and (iii) that supports heuristics in order to reduce the optimization rewriting phase.

1 Introduction

Many declarative query languages for object-oriented database management systems have been proposed in the last few years (e.g. [4, 5, 9]). Currently, a very serious concern is their optimization. Interesting techniques have been imported from other environments or developed for this context. However, although complementary, these techniques are often supported by distinct formalisms. The implementation of a query optimizer is thus still an awkward and delicate operation. Clearly, it is crucial to develop a general framework for object-oriented queries optimization. We are taking one important step in this direction by proposing an algebraic formalism that unifies the various rewriting techniques that have been developed in the object-oriented database context.

The essence of a query optimization is to find an execution plan that minimizes a cost function. An optimization process traditionally involves two deeply connected levels that are qualified as logical and physical. The logical level uses the semantic properties of the language in order to find expressions equivalent to the one given by the programmer (query rewriting). With a few notable exceptions, this is usually done in the relational context by applying equivalences to algebraic expressions. The physical level uses a cost model based on system informations to choose the best algorithm for the evaluation of a given expression. Among other techniques, the relational database community has produced many interesting join algorithms that make clever use of the main memory for minimizing the expensive input-output operation. Another aspect of query optimization concerns the search space for the best execution plan of a query. It can be very large and, in some cases, considering all the alternatives may be more expensive than just plainly executing the query. To solve this problem, different search strategies have been developed in the relational context.

Much effort has been done in order to adapt the techniques developed for the relational database systems to the object-oriented environment. For instance, a number of researchers have worked on algebraic query rewriting (e.g. [17, 6, 18], others have extended relational search strategy [16]. However, the characteristics of the object-oriented environment had lead to consider entirely new techniques.

First, objects are not flat tuples. An object may recursively refer to other objects. In this context, queries often involve pointer chasing operations through the different components of the database objects. This kind of operations may imply numerous input-output operations (I/O). One way to minimize I/O is to consider class extents in order to transform navigation into algebraic join operations [13, 15]. Another technique consists in having a specialized module in the optimizer

*A preliminary version of this paper, under the title *Towards a Unification of Rewrite Based Optimization Techniques for Object-Oriented Queries*, has appeared as a technical report for the Basic Research Action Fide, contract 3070 presented at the *Esprit Conference, Brussels, Belgium, November 1991.*

†Inria, B.P. 105, 78153 Le Chesnay Cedex, France

‡Université de Paris-Sud, LRI, 91405 Orsay Cedex

whose task is to avoid an object-at-a-time reading of the various components needed in the evaluation of the query [14].

Secondly, a component of an object may be a nested structure. This leads, for instance, to considering expressions denoting complex paths (e.g. employee.company.address.city.name) that may be duplicated. In this context, common subexpressions factorization is an issue that cannot be ignored.

Finally, object-oriented queries may contain method (i.e. function) calls. This yields the need to consider a preliminary phase to the optimization process which consists in finding the methods code (and eventually translates it to an algebraic formalism) [12]. One encounters then problems raised by the overloading of methods name which are common to object-oriented languages.

The goal of our work is not to introduce yet another technique for the optimization of object-oriented queries, but to integrate in a common simple framewok the different techniques that have been proposed. In this paper, the focus is on the logical aspect of query optimization. Although we do take into account indexes and objects clustering, we do not consider important aspects such as the cost model or the choice of a particular join algorithm. We provide a detailed analysis of the queries through a formalism (i) that unifies different rewriting techniques, (ii) that allows easy and exhaustive factorization of duplicated subqueries, and (iii) that supports heuristics in order to reduce the optimization rewriting phase.

There exist two main approaches to query rewriting in the object-oriented context: the algebraic approach [17, 6, 18] and the one, that we call the type based approach, which transform pointer chasing into join operations [13, 15]. We unify these two appoaches in one formalism. We show that it is just the lack of type information (membership to class extents) that makes it impossible to perform type based transformations using algebraic equivalences. Accordingly, we extend the algebraic approach by reducing and typing the complex expressions representing selection, projection and join criteria.

When these different techniques are combined, the number of possible expressions of a simple query is so large that it is fundamental to investigate heuristics to limit the search space for an equivalent expression, thereby reducing the rewriting phase. The heuristics we advocate rely on the knowledge of indexes [8] and objects placement policies [7]. Indexes and object placement policies can respectively be represented in our model by paths of length equal or greater than one and by trees. We have chosen for algebraic expressions a DAG representation on which they are easily mapped.

Evaluation algorithms often consider sequences of join-selection-projection operations as a whole. In order to obtain an exhaustive factorization of the subexpressions duplicated in a query, we associated a unique representation to each such sequence. By eliminating the boundaries between subexpressions belonging to distinct algebraic operations, this unique representation allows an easy and complete factorization.

The framework presented here was suggested by problems encountered while implementing the O_2Query optimizer [10]. It is at the basis of the second optimizer that is being implemented.

This paper is organized as follows. In the next section, we present an object-oriented data model, some systems characteristics and the languages we will be using in the paper. We explain our motivations and goals in Section 3. Section 4, 5 and 6 introduce our formalism in three stages. We conclude in Section 7.

2 Preliminaries

We assume that the reader is reasonably familiar with most of the concepts found in object-oriented models and systems.

The Data Model

There are about as many object-oriented data models as there are projects working on the subject. However, most of these models support the features proposed in [3]. The model we consider here is close to the O_2 data model [11] and is general enough so that many different ones can be mapped on it.

The model has classes and concrete types. Each class has a name, defines a structure and a behaviour. Classes obey the inheritance principle, they may have an extent, i.e. the set of all their instances. Their instances are called objects and respect the encapsulation and identity principles. A concrete type only defines a structure. It may be named or not. Its instances are called values and know neither encapsulation nor identity. A concrete type does not have an extent. Concrete type can also be found in the Exodus data model [9].

The next figure presents the classes of a partial toy schema for a travel agency database. We comment it briefly. In the representation, concrete type names are in lowercase (e.g. "activities") while class names are capitalized (e.g. "Country"). An object of class "Country" has two atomic attributes "name" and "continent" and an object attribute "capital". Class "Capital" is a subclass of "City". Set valued attributes are found in Classes "Informations" and "CV".

Classes and Types
Country: tuple (name: string, capital: Capital continent: string)
City: tuple (name: string, country: Country, info: Informations)
Capital is a City
Informations: tuple (hotels: Set(Hotel), day: activities, night: activities)
Destination: tuple (city: City, hotel: Hotel)
Employee: tuple (cv: CV, sales: set(Sale))
CV: tuple (name: string, bplace: birthplace)
Sale: tuple (dest: Destination, employee: Employee, amount: float)
activities: set (string)
birthplace: tuple (city: string, country: string)

Access path information

Optimization strategies are traditionally based on system informations provided by a database administrator (DBA). We are considering four kinds of informations that are found in part or totally in most object-oriented systems: inverse link between classes, class extents, indexes and clustering strategies.

Some systems, like Vbase [2], offer the possibility to maintain inverse links between two classes. For instance, the attribute "country" in class "Capital" can be defined by the DBA as being the inverse of attribute "capital" in class "Country".

Most of the object-oriented systems maintain class extents even if they are not always accessible to the programmers (e.g. in O_2 [11]). The extent of a class is the set containing all its instances. We have seen in the introduction, and we will come back to it in a more detailed manner in the next sections, that extents could be used to transform pointer chasing operations into algebraic operations. As such, extents are useful elements for an optimizer.

Indexes in relational systems involve only one relation. In object-oriented systems, structures are more complex and, as a consequence, one has to consider having indexes on path of length greater than one going through several classes. In [8], different kinds of index are proposed. In this paper, we will only consider *path indexes* but the way we are using them can easily be extended to others. A path index is a data structure which provides a direct backward link.

The last information that we consider concerns knowledge about the clustering of objects on disks. Object-oriented systems may offer different clustering possibilities, namely class, composition or random strategies. A class clustering consists in having all the instances of a class in one single file. A composition clustering groups, in one file, objects of a class along with one or more of their composite objects. With a random strategy, objects are stored in their order of creation on one unique object space. In O_2, the three clustering policies are available and expressed by *placement trees* [7]. Orion and Exodus offer class and composition strategies [13, 9].

The model that we propose for query optimization takes into account these different informations: inverse links, extents, indexes and clustering.

Languages

There are many declarative query languages for object-oriented database systems (e.g. [4, 5, 9]). These languages mainly differ on one point and, accordingly, can be partitionned into two classes. The first contains languages that only allow the selection of objects already present in the database. Languages of the second class are more powerful by the fact that they make it possible to generate new entities. We chose to express queries in the O_2Query language [4] which belongs to the second category.

O_2Query is a functional language. It is defined by a set of basic functions and a way of building new functions from these, through composition and iterators. The language iterators have been given an SQL-like syntax. The queries we will present do not use methods. This is why we have not defined any on the schema. We assume that a method invoked in a query has no side effects and thus that, as far as rewriting is concerned, a method call is comparable to a field selection. Since we do not yet consider accessing methods code or evaluating cost, the difference between the two operations would lay in their arity. We will see that our formalism is able to treat n-ary as well as unary operations. The information on a method possible side effects is not easy to obtain but we will not consider the subject in this paper.

There are different algebras proposed for object-oriented models (e.g. [17, 18]) and, as for the languages meant for end-users, they mainly differ on the fact that some do not generate new entities and some do. The Encore algebra [17] belongs to the second category. Our formalism relies on a similar algebra. The main operators are defined as follows:

- Select(A, λ a, p(a)) selects the elements "a" of set "A" such that "p(a)" holds. Predicate "p" is a combination of the O_2 query language basic boolean functions.

- Project(A, λ a, f(a)) applies function "f" to the elements of set "A". Function "f" is a combination of the O_2 query language basic functions (field selection, message sending, tuple construction, ...). This operator is a combination of the ENCORE "Image" and "Project" operators.

- Join(A, B, λ a,b, p(a,b)) joins sets "A" and "B" according to predicate "p". If no predicate is given, the join operation is equivalent to a Cartesian product. In the relational algebra, the join operation takes two set of tuples and returns a set of tuples. Because of their data model, object algebras work differently. The join operation takes two sets of anything and returns a set of tuples. Thus, each join operation adds a level of tuple nesting. This unfortunately means the loss of the join associativity property as known in the relational model. It is a real handicap. Therefore, we propose the following *ad hoc* variation.

  – If the elements of set "A" are tuple **values** of type $<a_1:e_{a_1}, a_2:e_{a_2}, ..., a_n:e_{a_n}>$ and the elements of set "B" are not, then the result of the join operation is a set of tuple **values** $<a_1:e_{a_1}, a_2:e_{a_2}, ..., a_n:e_{a_n}, b:e_b>$ where e_b represents an element of set "B".

  – If the elements of set "A" and "B" are tuple **values** of type $<a_1:e_{a_1}, a_2:e_{a_2}, ..., a_n:e_{a_n}>$ and $<b_1:e_{b_1}, b_2:e_{b_2}, ..., b_m:e_{b_m}>$, then the result of the join operation is a set of tuple **values** $<a_1:e_{a_1}, a_2:e_{a_2}, ..., a_n:e_{a_n}, b_1:e_{b_1}, b_2:e_{b_2}, ..., b_m:e_{b_m}>$.

  – If the elements of set "A" and "B" are not tuple **values**, then the result of the join operation is a set of tuple **values** $<a:e_a, b:e_b>$ where e_a and e_b respectively represent elements of set "A" and set "B".

This set of operators plus standard set operators can support most of the query languages for object-oriented databases. They are defined on and return set values in contrast with the Encore operators that are defined on and return set objects. Thus, the algebra we consider can be compared to complex objects algebras (e.g. [1]. The fact that we do not generate object identifiers eliminates the distinction that is made in the Encore algebra between identity and structural equivalences.

3 Motivations and Goals

Many interesting techniques have been developed for the optimization of object-oriented queries. However, although complementary, these techiques are often supported by distinct formalisms. Our goal is to unify in a single framework these different techniques. As a first step, we focus essentially on the logical aspect of query optimization. We consider the integration of two kinds of query rewriting: one that uses class extents in order to transform pointer chasing operation into algebraic joins and the other, more traditional, that is based on algebraic equivalences. We also want to be able to factorize the constant or duplicated subexpressions of a query. Finally, since the combination of the two rewriting techniques will make it possible to return a large number of expressions equivalent to a single query, we are also interested in finding heuristics to reduce the search space for an equivalent expression. We will now give more details about these different points.

The authors of [13] propose a graphical representation of queries and tree traversal algorithms that allows rewritings based on types. This optimization concerns a distributed system but is also relevant in a centralized one. The queries are those of the Orion language first release [5] that can be translated by a selection operation in an object algebra. As a matter of fact, and this is its main drawback, the Orion technique does not concern other algebraic operations. This drawback has been overcome in [15].

Let us consider the following query expressed in the O_2Query language:

Query Q1 What are the African capitals where one can scuba dive at day and swim at midnight?

select	c
from c	in Capital
where	c.country.continent = "Africa" and
	"scuba diving" in c.info.day and
	"midnight bathing" in c.info.night

□

Its corresponding Orion representation is given on Figure 1.

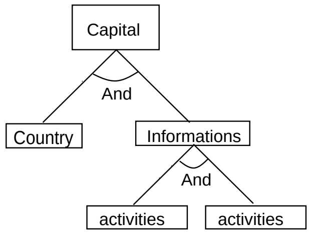

Figure 1: Orion representation of the query

The tree describes the complex types of the selection criteria. In the Orion data model, to a complex type corresponds an extent. Accordingly, to a node we may associate a set.

The tree is first reduced through decomposition into *clusters*. A cluster is a subtree whose root is an attribute node. The tree on Figure 1 may be reduced in two ways. We may consider the tree as a whole or build a cluster whose root is the "Informations" node.

There are three methods for evaluating a cluster The forward traversal starts at the root node of a cluster. It consists of a projection on the children attributes, an evaluation of the qualified children that, then, allows the qualification of their parent. This can be done algebraically (projection, selection, semi-join) or instance per instance. The backward traversal starts at the cluster leaves. The qualified attributes are evaluated through a selection. Then, a semi join returns the qualified parents. The mixed traversal combines the two previous methods. Some links are evaluated in forward traversal and others in backward traversals. For instance, in Figure 1 the query may be evaluated through a backward traversal from node "Country" to node "Capital" and then through a forward traversal toward the node "Informations".

These three kinds of traversals lead to a great number of possible evaluations of the simple Query Q1. It goes from a naïve instance per instance forward traversal to more complex evaluations involving joins.

We next consider the algebra based rewriting technique. In algebraic terms, the query is the following selection:

Algebraic expression E1

select (Capital, λ c
  (c.country.continent = "Africa" and
  "scuba diving" in c.info.day and
  "midnight bathing" in c.info.night))
□

On such a simple query, the only possible transformation is a partitioning of the selection operation. For instance, we can rewrite the query in the following manner:

Algebraic expression E2

Select (Select (Capital, λ c
    (c.country.continent = "Africa"))
  λ c ("scuba diving" in c.info.day and
    "midnight bathing" in c.info.night))
□

Assuming that there is an index on the path "country.continent" of the capitals, this rewriting would emphasize the use of that index.

However, there seems to be no way to generate, from this algebraic expression, the join operations that were detected by the Orion technique. Neither is it possible to emphasize the use one could make of an index on the path "continent" of the "Country" extent and of an inverse property on the attribute "country" of class Capital. In other words, it seems that we cannot transform Expression E1 into the following expression, that first selects the African countries and work from then on to find the appropriate capitals:

Algebraic expression E3

Select (Project (Select (Country, λ c
      (c.continent = "Africa"))
    λ c (c.capital))
  λ c ("scuba diving" in c.info.day and
    "midnight bathing" in c.info.night))
□

This use of a partial index and of an inverse function is possible with the type based rewriting technique. It corresponds in Orion to a mixed traversal from node "Country" to node "Capital" and from then on to the two nodes "ativities".

On the other hand, the algebraic approach is not limited to the rewriting of selection operations whereas the type based rewriting technique is. We will illustrate this with an example that figures a join operation that is transformed into a selection operation. This kind of transformation, although converse to the Orion ones, is not considered in the type based rewriting technique.

Let us consider the following example:

Query Q2 Find the employees who have sold trips to their city of birth.

select e
from e in Employee, s in e.sales
where e.cv.bplace.city = s.dest.city.name
□

The query algebraic translation is the following:

Algebraic expression E4

Project (Select (Join (Employee, Sale, λ e, λ s
      (s in e.sales))
    λt (t.e.cv.bplace.city=t.s.dest.city.name))
  λ t (t.e)))
□

Now, let us consider the two following equivalences.

Equivalence Eq1

Select (Join (A, B, λ a, b p(a,b)) λ t p_s(t.a,t.b)) ≡
Join (A, B, λ a, b (p(a,b) and p_s(a,b))) □

Equivalence Eq2

Project (Join(A, B, λ a, b (p(a, b))) λ t (t.a)) ≡
Select(A, λ a, ∃ b (b in Bs and p(a, b))) □

Equivalence Eq2 only applies if we do not consider duplicates in the resulting set. Using this two equivalences, we can eliminate the query join operation and consider instead the following simple selection:

Algebraic expression E5

Select(Employee, λ e, $\exists$ s (s in e.sales[1] and
e.cv.bplace.city = s.dest.city.name))

□

The gain brought by this transformation is that we do not have to consider all the sales of a given employee but, instead, we stop at the first sale that validates the predicate.

Before we go further, we would like to point out a problem one has to consider when studying query optimization in a object-oriented data model where some types do not have an extent. In Query Q2, the definition of Variable "s" depends on Variable "e". In the corresponding join operation, Variable "s" is defined on its membership in the "Sale" extent and the variables dependence is translated with the predicate "(s in e.sales)". Now, supposing that the type associated with variable "s" did not have an extent, this would appear to forbid an algebraic translation. A solution to this problem is to consider two kinds of extents. Extents of the first kind are maintained by the database system while extent of the second are not but could be evaluated at a huge cost by scanning the database. We call "virtual extents" the extents belonging to the second category. In the data model we are considering, concrete types extents are virtual extents. The rules that manage these unordinary sets are simple enough. Select, projection, union and difference operations are not defined on virtual sets. Join operations between two virtual sets are not considered. One virtual set is accepted in a join operation if the join condition contains a membership test on the variable associated to the virtual set. For instance, the following join expression is acceptable:

Algebraic expression E6

Join (Informations, activities, λ i, a, (a in i.day)) □

The type based rewriting technique combined to the algebraic formalism will make it possible to return a large number of expressions equivalent to a single query. In the environment we are considering, the knowledge of placement policies makes it possible to eliminate some of them as not relevant. For instance, let us consider again the query represented on Figure 1 and let us suppose that the database administrator has specified the following storage policy:

```
create group on Capital:(info (day, night))
```

This means that all objects of class "Capital" should be stored along with their "info" component and followed by this component attribute values. In that case, and in the absence of appropriate indexes, we know that it will be more appropriate to evaluate the right part of the tree instance per instance in a forward manner rather than performing a join between the "capital" and "Informations" extent. For the same reasons, to partition the two selection conditions concerning the info attribute should not be considered.

Now, we will study the third technique, the factorization of common query subexpressions.

Let us consider the following query:

Query Q3 Find the employees who have sold trips to their city of birth. Give their names and cities of birth. The trips must be special offers or have prices greater or equal to $20,000.

```
select tuple(name: e.cv.name, city: e.cv.bplace.city)
from   e in Employee, s in e.sales
where e.cv.bplace.city = s.dest.city.name and
        (s.dest = special_offer or s.amount >= 20000)
```

□

This query can be translated into the following algebraic expression:

Algebraic expression E7

Project (Select (Join (Employee, Sale, λ e, s
(s in e.sales))
λ t (t.e.cv.bplace.city=t.s.dest.city.name
and (t.s.dest = special_offer or
t.s.amount>=20000)))
λ t (tuple(t.e.cv.name, t.e.cv.bplace.city)))

□

Let us now emphasize anomalies that the two approaches we have studied so far are powerless to solve.

The operation "e.cv.bplace.city" is performed twice on the qualified employees, once to test a selection condition and the second time to perform the projection operation. Another point is that this operation is evaluated for every pair (employee, sale) while it only concerns employees. One must remember that in an object oriented environment, a simple field selection may cause a page fault. Thus, it might be a gain to factorize all possible operations especially if this can be done at a lesser cost. However, it seems that none of the two approaches we previously studied can provide an appropriate treatment of these anomalies.

We plan to overcome this drawback.

To Summarize

We are considering a formalism for the logical layer of an optimizer. We want it (i) to subsume the type based rewriting technique and the algebraic approach, (ii) to support informations on objects placement policies and indexes in order to reduce the rewriting phase and (iii) to allow easy and exhaustive factorization of common subexpressions and to emphasize subexpressions depending of one variable in a multi-variables query.

We will present informally our solution in three stages, each one corresponding to one of our three goals. A formal definition can be found in [10].

4 A Simple Idea: a Typed Algebra

To each possible traversal of the Orion tree representing a query, one may associate an algebraic expression. If we consider the query represented on Figure 1, a forward instance per instance traversal corresponds to Expression E1, a mixed traversal starting from node "Country" going backward to node "Capital" and then forward, instance per instance, from node "Capital" to the nodes "activities" may correspond to Expression E3.

However, we have seen that, using algebraic equivalences, it seemed impossible to go from the first algebraic expression to the second. Neither was it possible to generate joins operations from a simple selection. These limitations are due to the lack of a good typing information.

To overcome this shortcoming, a simple idea is to consider all the elementary operations involved in an algebraic expression and type them.

We illustrate this on Expression E1 that is appropriately expanded to incorporate the necessary typing:

Algebraic expression E8

Select (Capital, λ c_a, $\exists$ c_{ou}, c_{ont}, i, a_d, a_n,
  (c_{ou} in Country and c_{ont} in string and
    i in Informations and a_d in activities and
    a_n in activities and
  c_{ou} = c_a.country and c_{ont} = c_{ou}.continent
    and i = c_a.info and a_d = i.day
    and a_n = i.night and
  c_{ont} = "Africa" and "scuba diving" in a_d
    and "midnight bathing" in a_n))

□

On the first line, we associated a variable to each elementary operation. On the second and third lines, we specified the variables types by their membership in virtual or real extents. The next two lines define the links between variables and the last line expresses the initial selection conditions.

In more formal terms, the selection operation has been expanded in the following manner:

Select(A, λ a, $\exists$ v_1, v_2, ..., v_n
  (p_{type}(v_1, v_2, ..., v_n) and
  p_{def}(a, v_1, v_2, ..., v_n) and
  p_{sel}(a, v_1, v_2, ..., v_n)))

Predicate p_{type} specifies the variables types and Predicate p_{def} defines the links between the different variables. It is important to note that to define these links is to express all the elementary operations one has to perform. Predicate p_{sel} is the initial selection condition.

We note that every variable represents a different operation. The expression "c.info" that figured twice in the original selection has been factorized and appears only once in this new translation. However, this factorization does not give us entire satisfaction since it is just local to an operation. We will come back later to this problem.

Now, let us consider a new algebraic equivalence.

Equivalence Eq3

Select(S1, λ s s.a in S2) $\equiv$ Project(S2, λ a inverse(a))

□

If we apply this equivalence to Expression E8, we may generate a typed version of Expression E3. We previously used equivalence Eq2 to transform a join operation into a selection. It can also be used the other way and allows us to generate as many joins as possible between the various extents. For instance, we may generate the following expression:

Algebraic expression E9

S_{join} $\leftarrow$ (Join (Capital, Informations, λ c_a, i,
  $\exists$ c_{ou}, c_{ont}, a_d, a_n ,
  (c_{ou} in Country and c_{ont} in string and
    a_d in activities and a_n in activities
  c_{ou}=c_a.country and c_{ont}=c_{ou}.continent
    and i=c_a.info and a_d=i.day
    and a_n=i.night and
  c_{ont} = "Africa" and "scuba diving" in a_d
    and "midnight bathing" in a_n))
Result $\leftarrow$ Project(S_{join}, λ t, $\exists$ c_a,
  (c_a in Capital and
  c_a = $t.c_a$),
  c_a))

□

This expression consists of a join operation, whose condition is the initial selection condition, and of a projection. Other equivalences can be applied to this expression to introduce new selection or joins operations.

We notice that the typing we performed on the selection operation have also been performed on the join and projection operations.

The Encore algebraic equivalences are adapted to this new formalism in a straightforward manner as shown in [10].

At this stage we have reached our first goal. By introducing typing and intermediary variables in the algebraic expressions, we made it possible to use equivalences that could not be applied otherwise (e.g. equivalences Eq2 and Eq3 on expression E1) and that generate expressions equivalent to the Orion tree traversals. It is obvious that all the other algebraic equivalences are still applicable. Accordingly, we have a formalism that subsumes the Orion technique and the traditional algebraic formalism.

Another major goal was to support informations on object placement policies and indexes in order to reduce the rewriting phase. Since these informations are represented by trees and paths, we chose a DAG representation for the algebra.

5 Graphical Representation of a typed Algebra

Let us first study informally the graphical representation of Expression E8 given on Figure 2.

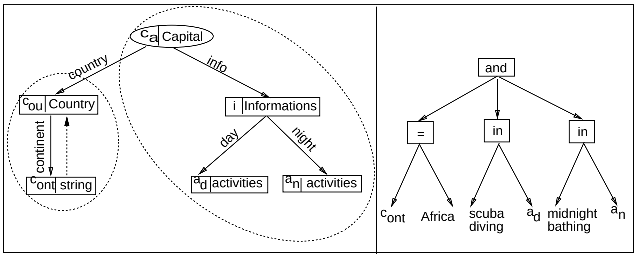

Figure 2: An Adorned Graphical Representation of Expression E8

For the moment, we will ignore the dashed lines represented on this figure.

The selection operation consists of (i) a graph figuring the selection variables, their types, the way they are linked and (ii) a tree representing the selection condition.

The graph has two different kinds of node. The oval node represents the set on which the selection is made and its associated variable (here c_a: Capital). Each rectangular node represents an intermediary variable (c_{ou}, c_{ont}, i, a_d, a_n) and its type (p_{type} in the textual representation).

The edges represent the operations that link the variables (p_{def} in the textual representation). There cannot be two edges representing the same operation starting from the same node. This guaranties an exhaustive local factorization and at a lesser cost since the number of edges starting from one node is limited by the type of the node. The operations of the example are unary. However, as we will see later, we can also represent n-ary operations (method calls, tuple construction, ...). An n-ary operation is represented by "n" numbered edges having same destination.

The initial selection predicate (p_{sel}) is represented by a tree whose leaves are the variables or the constants on which the predicate is defined.

Now, let us consider the dashed lines that represent system informations. On the figure, areas bordered by a dashed line indicate placement trees and the dashed edge represents an index. These informations, as we will illustrate it now, are used to limit the rewriting phase.

The first rule to limit rewriting is given below.

Rule R1
In the absence of appropriate indexes, it is preferable not to introduce joins on nodes that are in a same dashed area.
□

For instance, Nodes c_a, i, a_d and a_n are in a same area that does not contain any index edge. Thus, it is better not to introduce a join between the "Capital" extent of Node c_a and the "Informations" extent of Node i. Accordingly, Expression E9 cannot be generated from this DAG representation. This restriction can easily be understood by the fact that, according to the placement algorithm, the values corresponding to the nodes figuring in a same area are stored in sequential order in the same file.

Another rule concerning placement policies is the following:

Rule R2
In the absence of appropriate indexes, it is better not to partition a selection in two if the corresponding node are in a same dashed area.
□

For instance, we will not allow a partitioning of the selection operation that would cut the tree formed by Nodes i, a_d and a_n into two.

If we do not consider the ordering of the conditions expressed in a selection, rules R1 and R2 leave us with only four possible expressions of the query. The one represented on the graph, Expression E3, plus two others that we do not give for lack of space.

We have seen some ways to use placement policies to reduce rewriting. This can also be done by simply using index informations. Details on this can be found in [10].

We now summarize what has been achieved. User queries are transformed into typed algebraic operations that are represented graphically and that support informations on object placement policies and indexes. The transformations we perform are based on algebraic equivalences and limited by rules concerning object placement policies and indexes. In other words, our optimizer rewriting rules are composed of (i) an *algebraic condition* that specifies the structure of the expression before its transformation, (ii) a *system condition* that prohibits transformation when the objects placement policy and indexes context is not right and

(iii) an *algebraic transformation* that specifies the resulting algebraic expression.

At this stage we have reached two out of three goals. We will now see how to achieve global factorization.

6 A Global Representation for a Global Factorization

Sequences of join, selection and projection operations are liable to be evaluated by a same algorithm. For instance, Expression E7 can be evaluated by a nested loop algorithm. In these sequences, one may find subexpressions common to distinct algebraic operations (e.g. "e.cv.bplace.city" common to the selection and projection operations of Expression E7). The type based and algebra based rewriting techniques are powerless to detect and factorize these expressions because they consider separately each algebraic operation.

To solve this problem, a simple idea consists in having a global representation for each sequence of algebraic expressions that may be evaluated as a whole. In this global representation, the boundaries between subexpressions belonging to distinct algebraic operations are invisible. This allows exhaustive factorization whereas the previous approaches only considered local factorization.

We will illustrate our meaning by considering the global representation of Expression E7 given on figure 3.

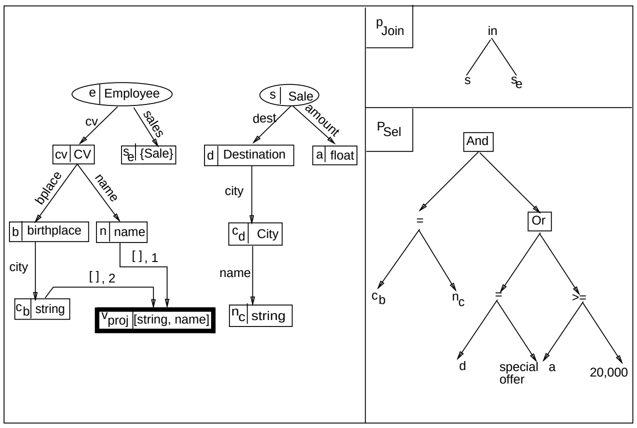

Figure 3: A graphical representation of Expression E7

For clarity, we have not adorned the representation. It figures the three algebraic operations of join, selection and projection. It consists of (i) two subgraphs whose roots represent the sets on which the join operation is performed and whose other nodes figure intermediary variables representing elementary operations, their types, the way they are linked, (ii) a tree representing the join condition and (iii) a tree representing the selection condition. The variable representing the projection operation is in a bold rectangle (v_{proj}). In this example, the two subgraphs rooted at Nodes "Employee" and "Sale" are disjoint. However, this is not always the case.

The graph has two different kinds of nodes. The oval nodes represent the sets on which the operations are defined and their associated variables (e, s) and the rectangular nodes represent the intermediary variables and their types (cv, b, c_b, n, v_{proj}, s_e, d, c_d, n_c, a). The edges represent the operations that link the variables. One may notice the representation of an n-ary operation. The function that constructs a tuple having two attributes is represented by two labelled edges toward variable v_{proj}.

This grouped representation of three sequential algebraic operations allows an exhaustive factorization of the query common subexpressions that would not have been possible by considering each operation separately. For instance, the subexpression "e.cv.bplace.city", that is represented by the intermediary variable c_b, is common to the selection and projection operations and has been factorized. One can also notice two disjoint subgraphs. The first that includes Nodes e, cv, b, c_b, n and v_{proj} concerns the operations on the "Employee" objects. The second, that includes Nodes s, d, c_d, n_c and a concerns the operations on the "Sale" objects. These disjoint subgraphs indicate that these operations do not have to be evaluated for every pair (employee, sale) resulting of the join operation and that it would be a gain to push them out of the join operation.

So, once again, let us summarize what has been achieved.

User queries are transformed into typed algebraic expressions that are represented graphically. The typed algebra is now limited to three operators: union, intersection and an operator representing three ordered and sequential operations of join-selection-projection that we call JSP.

The textual expression of this operator is as follows:

JSP (A, B, $\lambda\ a, b, \exists\ v_{proj},\ v_1,\ v_2, ..., v_n$
$(\mathrm{p}_{type}(v_{proj},\ v_1,\ v_2,\ ...,\ v_n)$ and
$\mathrm{p}_{def}(a,\ b,\ v_{proj},\ v_1,\ v_2,\ ...,\ v_n)$ and
$\mathrm{p}_{join}(a,\ b,\ v_{proj},\ v_1,\ v_2,\ ...,\ v_n)))$ and
$\mathrm{p}_{sel}(a,\ b,\ v_{proj},\ v_1,\ v_2,\ ...,\ v_n),$
$v_{proj}))$

Variables a and b are defined on the two sets we are considering. Variables v_{proj}, v_1, $v_2, ..., v_n$ are intermediary variables representing the join, selection and projection elementary operations. Predicate p_{type} defines the intermediary variables types. Predicate p_{join} is the join predicate and predicate p_{sel} is the selection predicate. Variable v_{proj} represents the projection operation.

The transformation from a select-from-where user query to a graphical typed algebra expression is straightforward. Once this transformation is made we have succeeded in a *global factorization* that we will keep through the rewriting process.

As we specified it, rewriting rules comport three parts. Two that are traditional (*algebraic condition* and *algebraic transformation*) plus one that concerns system informations and that will be used to *reduce the rewriting phase*.

As we have seen, the intermediary variables we included in the algebraic expressions and the appropriate type informations give us *the combined power of algebra based and type based rewriting techniques*.

So, finally, we have reached the three goals we defined in section 3.

7 Conclusion

In this paper, we have presented a formalism that covers the rewriting phase of a query optimizer for an object-oriented database system. It consists of a DAG typed algebra that subsumes the type based and the traditional algebra based rewriting techniques. It supports informations on objects placement policies and indexes that are used to reduce the rewriting phase. Finally, it also allows a simple and exhaustive factorization of the duplicated and constant subexpressions of a query.

References

[1] S. Abiteboul and C. Beeri. On the Power of Languages for the Manipulation of Complex Objects. Technical report, INRIA and the department of computer science of the Hebrew University of Israel, 1987.

[2] T. Andrews and C. Harris. Combining Language and Database Advances in an Object-Oriented Development Environment. In *Proc. OOPSLA, Orlando, Florida, USA*, October 1987.

[3] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. The Object-Oriented System Manifesto. In *Proc. DOOD, Kyoto, Japan*, December 1989.

[4] F. Bancilhon, S. Cluet, and C. Delobel. Query Languages for Object-Oriented Database Systems: the O2 Proposal. In *Proc. DBPL, Salishan Lodge, Oregon, USA*, June 1989.

[5] J. Banerjee, W. Kim, and K. Kim. Queries in Object-Oriented Databases. Technical Report DB 188-87, MCC, Austin, Texas, USA, June 1987.

[6] C. Beeri and Y. Kornatzky. Algebraic Optimization of Object-Oriented Query Languages. In *Proc. ICDT, Paris, France*, 1990.

[7] V. Benzaken and C. Delobel. Enhancing Performance in a Persistent Object Store: Clustering Strategies in O_2. In *Proc. POMS, Martha's Vineyard, Massachusetts, USA*, September 1990.

[8] E. Bertino and W. Kim. Indexing Techniques for Queries on Nested Objects. *IEEE Transaction Knowledge and Date Engineering*, January 1989.

[9] M. Carey, D. DeWitt, and S. Vandenberg. A Data Model and Query Language for EXODUS. In *Proc. SIGMOD, Chicago, Illinois, USA*, 1988.

[10] S. Cluet. *Langages et Optimisation de requêtes pour Systèmes de Gestion de Base de données orienté-objet*. PhD thesis, Université de Paris-Sud, 1991.

[11] O. Deux et al. The Story of O2. *IEEE Transaction on Knowledge and Date Engineering*, 2(1), March 1990.

[12] G. Graefe and D. Maier. Query Optimization in Object-Oriented Database Management Systems with Encapsulated Behaviour. Technical report, Oregon Graduate Center, 1989.

[13] P. Jenq, D. Woelk, W. Kim, and W. Lee. Query Processing in Distributed ORION. In *proc. EDBT, Venice, Italy*, March 1990.

[14] T. Keller, G. Graefe, and D. Maier. Efficient Assembly of Complex Objects. In *Proc. SIGMOD, Denver, Colorado, USA*, 1991.

[15] A. Kemper and G. Moerkotte. Advanced Query Processing in Object Bases Using Access Support Relations. In *proc. VLDB, Brisbane, Australy*, 1990.

[16] R. Lanzelotte and P. Valduriez. Extending the Search Strategy in a Query Optimizer. In *Proc. VLDB, Barcelona, Spain*, 1991.

[17] G. Shaw and S. Zdonik. An Object-Oriented Query Algebra. In *Proc. DBPL, Salishan Lodge, Oregon, USA*, June 1989.

[18] D. Straube and T. Özsu. Queries and Query Processing in Object-Oriented Database Systems. Technical report, Department of computing science, university of Alberta, Edmonton, Alberta, Canada, 1990.