Evaluating Queries with Generalized Path Expressions

Vassilis Christophides
I.N.R.I.A.
78153 Le Chesnay, Cedex
France
Vassilis.Christophides@inria.fr

Sophie Cluet
I.N.R.I.A.
78153 Le Chesnay, Cedex
France
Sophie.Cluet@inria.fr

Guido Moerkotte
Lehrstuhl für Informatik III
Ahornstr. 55, 52074 Aachen
Germany
moer@gom.informatik.rwth-aachen.de

Abstract

In the past few years, query languages featuring generalized path expressions have been proposed. These languages allow the interrogation of both data and structure. They are powerful and essential for a number of applications. However, until now, their evaluation has relied on a rather naive and inefficient algorithm.

In this paper, we extend an object algebra with two new operators and present some interesting rewriting techniques for queries featuring generalized path expressions. We also show how a query optimizer can integrate the new techniques.

1 Introduction

In the past few years there has been a growing interest in query languages featuring generalized path expressions (GPE) [BRG88, KKS92, CACS94, QRS+95]. With these languages, one may issue queries on data without exact knowledge of its structure. A GPE queries data and structure at the same time. Although very useful for standard database applications, these languages are vital for new applications dedicated, for instance, to (semi)structured documents.

There have been some proposals for evaluating queries with generalized path expressions [BRG88, CACS94]. However, these proposals are rather naive and inefficient. In this paper, we present an algebraic approach to deal with GPEs in the object-oriented context. Therefore, we extend an object algebra by two new operators and then demonstrate the advantages of this clean and flexible approach.

So far, the technique for evaluating GPEs has been to find structure (type) information, rewrite the query accordingly, using unions or disjuncts, and, from then on, perform standard optimization. The main drawbacks of this technique are 1) exponential optimizer input (as many disjuncts/unions as possible paths), 2) no flexibility (fixed treatment), 3) many redundancies and 4) too many intermediate results.

In contrast, we extend an object algebra with two operators: one dealing with paths at the schema level, and one dealing with paths at the instance level. This approach remedies all of the above deficiencies. Further, we propose a data representation that considerably reduces the size of intermediate results.

The paper is organized as follows. After discussing related literature in Section 2, we justify our choices and introduce the extended algebra in Section 3. In Section 4, we illustrate our technique through some examples. Section 5 shows how our technique can be integrated into an optimizer. We conclude in Section 6.

2 State of the Art

Generalized path expressions (GPE) are very useful primitives that allow data as well as their structure to be uniformly queried. Sometimes, the structure of the data is imposed by the schema; sometimes some flexibility is left and there is no schema fixed in advance. Indeed, there are different motivations for introducing GPEs into query languages. In [BRG88, CACS94], GPEs were used as a means to provide better tools for querying documents stored in an object base. The particularity of that kind of application is that users are interested in data at various granularity levels. One may want to view a whole section as a text, or to see the different paragraphs involved in a section. The schema being fixed, this means that a user may want to ignore some parts of it. Thus the language must allow interrogation with partial schema knowledge. In [KKS92], GPEs were used to deal with the fact that, in object-oriented (oo) systems, some information is captured by the schema and, thus, it is important to be able to query the schema. Another motivation for GPEs is related to the interrogation of semistructured data [QRS+95] which has no absolute schema fixed in advance and its structure may be irregular or incomplete. In a similar

select struct(person:p,wine:w)
from Person{p}, Wine{w}
where p.name = w.chateau

select struct(person:p,wine:w)
from Person{p}, Wine{w}
where p.name = w.age

select struct(person:p,wine:w)
from Person{p}, Wine{w}
where p.name = w.country

select struct(person:p,wine:w)
from Person{p}, Wine{w}
where p.age = w.chateau

select struct(person:p,wine:w)
from Person{p}, Wine{w}
where p.age = w.age

select struct(person:p,wine:w)
from Person{p}, Wine{w}
where p.age = w.country

select struct(person:p,wine:w)
from Person{p}, Wine{w}
where p.car = w.chateau

select struct(person:p,wine:w)
from Person{p}, Wine{w}
where p.car = w.age

select struct(person:p,wine:w)
from Person{p}, Wine{w}
where p.car = w.country

Figure 1: An example of the naive query evaluation

direction, we believe that the current trend towards the interoperability of heterogeneous systems is yet another motivation for being able to evaluate efficiently GPEs. It is realistic to imagine future database systems as the receivers of huge amounts of information either stored locally or viewed from distant sites. In this framework, GPEs will be useful either because they offer shortcuts in the expression of a query or because of incomplete knowledge of the schema.

Generalized Path Expressions At this point, the reader may be interested in seeing a GPE. Let us consider the following example featuring two GPEs in the **from** clause. The language is an extension of the OQL language and was introduced in [CACS94].

select struct(person: p, wine: w)
from Person{p}.A, Wine{w}.B
where p.A = w.B

The query pairs Persons together with appropriate birthday presents. All birthday presents are taken from our wine cellar. To make the present special for the friends, we select those that must have something in common with the friend. In this query, p and w are data variables, that iterate over all members of *Person* and *Wine*, respectively. A and B are attribute variables, that iterate over all possible attributes. Later, we will also use path variables, that iterate over all possible paths[1]. Within the example, the ranges of A and B are restricted to the attributes of *Person* and *Wine*. The semantics of the query can be thought of as being the following (for a formal semantics see [CACS94]). For each possible attribute binding the query is evaluated. If a type error occurs, the binding does not result in any qualifying tuples. For all "correct" bindings (wrt. typing) of the attribute variables, the results of the instantiated queries are unioned to give the final result. If no "correct" binding exists, the query results in an empty set.

[1] We assume that the reader has an intuitive understanding of a path within the object base context.

Naive Query Evaluation This semantics results in a first evaluation strategy of queries with attribute and path variables:

1. look for all possible instantiations of attribute and path variables;
2. replace the attribute and path variables by their instantiations;
3. eliminate the not well-typed alternatives;
4. union the remaining instantiated queries and evaluate the resulting query.

Let us demonstrate this with the example query above and the following schema:

Person:[name:string; age:int; cars:{Car}];
Wine:[chateau:string, age:int, country:Country];

For *Person* and *Wine*, we have three attributes each. This results in nine possible instantiations given in Figure 1. All but two of these instantiations are ill-typed. The union of the two well-typed instantiations results in a regular query, that is, one without attribute (or path) variables:

select struct(person:p,wine:w)
from Person{p}, Wine{w}
where p.name = w.chateau
union
select struct(person:p,wine:w)
from Person{p}, Wine{w}
where p.age = w.age

The evaluation of this query, results in the answer to the original query. Note that:

- the number of subqueries explodes exponentially in
  a) the number of attributes per type involved in the query

b) the number of attribute variables

- if a distinct is used in the original query, expensive duplicate elimination becomes necessary.

The situation is even worse if path variables are involved since the number of possible paths is far larger than the number of attributes. Therefore, we look for alternatives.

Another possibility, proposed in [BRG88], is to use the Boolean connector *or* instead of a union[2]. For the example query, the result would be:

select struct(person:p,wine:w)
from Person{p}, Wine{w}
where (p.name = w.chateau **or**
p.age = w.age)

Note that this approach has a slightly different semantics: an entry satisfying all conditions will be selected only once. Anyway, the combinatorial explosion of possibilities is present in this approach as well. Furthermore, queries with disjunctions are often transformed into disjunctive normal form and then the disjunctions are translated into unions. This standard approach would be prohibitively expensive.

One may think that another solution for evaluating GPEs could be to transform GPE queries into standard queries over data and meta-schema. There are two reasons why we do not favor this solution. The first is that GPEs involving path variables imply some recursion over the meta-schema. Optimization of recursive queries is tricky and we'd rather not have to deal with it. The second reason is that meta-schemas are system dependent. Thus, this approach cannot be universal and a solution would have to be defined for each specific system/meta-schema.

3 Extending the Object Algebra

We believe that the solution lies in an algebraic treatment of GPEs. By extending an object algebra in an appropriate fashion, we will give control of the GPEs to the optimizer. This will result in more evaluation alternatives and will allow the creation and use of adequate data structures.

The idea of the naive evaluation strategy is to first perform some sort of schema lookup to obtain all the possible instantiations of path and attribute variables and then to restrict the instantiations of the query to those which result in well-typed queries. The valid instantiations are unioned and the resulting expression is given to an ordinary query optimizer. Obviously, there exist two separate main phases: schema lookup and type inference on the one hand, and query optimization and evaluation on the other hand.

[2]Actually, the method was proposed only for a very limited query language, but it is easy to generalize it.

Notice that if we consider real applications, the schema itself can be large and the naive evaluation may result in an exponential input for the optimizer. Hence, one of our goals is to apply query optimization techniques (i.e., factorization) for paths also to the schema lookup phase. Further, we want to avoid the separation into two phases, since, as we will see, there exist situations when simple object base lookups can restrict the number of possible paths enormously, without incurring additional costs.

Hence, our goal is to integrate schema lookup and object base lookup, and to be able to apply optimization techniques in a homogeneous fashion to both lookups. For this, we extend the algebra of [CM95a] with two new operators that instantiate GPEs from a schema and a data perspective. We start by a short presentation of the algebra before introducing the new operators along with data structures for their implementation. The next section will show how they can be used advantageously.

3.1 The Core Algebra

We assume standard knowledge of object-oriented data models. The algebra is an extension of the GOM algebra [KM93, KMP93]. Its main characteristic is that — with the exception of the map operator — it is defined on sets of tuples. This guarantees some nice properties among which is the associativity of the join operator.

We now present the algebraic operators that will be used in the following. Other operators and equivalences can be found in [CM95a].

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 dereferencing, method and function application. Also, our translation process pushes all nesting into map operators. The first definition corresponds to the standard map [KM93] or materialize [BMG93] operator. The second definition is just a shorthand for a map with tuple construction.

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

In the definitions, the e_i's denote both expressions (on the left hand side) and their evaluation (on the right hand side). Note that the oo map operator obviates the need for a relational projection.

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

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

Join Operations The algebra features different join operators. We present two. The first is called *d-join* ($< \cdot >$). This is a join between two sets, where the evaluation of the second set may depend 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 previously defined variable.

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

where $\circ$ represents tuple concatenation. Whenever possible, d-joins are rewritten into standard joins.

$$e_1 \bowtie_p e_2 \quad = \quad \{y \circ x | y \in e_1, x \in e_2, p(y, x)\}$$

The operator signatures are given below.

$$\begin{array}{rcl} Map_f & : & \{\tau_1\} \rightarrow \{\tau_2\} \\ & & \text{if } f : \tau_1 \rightarrow \tau_2 \\ \sigma_p & : & \{\tau\} \rightarrow \{\tau\} \\ & & \text{if } p : \tau \rightarrow Bool \\ \bowtie_p & : & \{\tau_1\}, \{\tau_2\} \rightarrow \{\tau_1 \circ \tau_2\} \\ & & \text{if } \tau_i \leq [], p : \tau_1, \tau_2 \rightarrow Bool \\ & & \mathcal{A}(\tau_1) \cap \mathcal{A}(\tau_2) = \emptyset \\ < \cdot > & : & \{\tau_1\} || \{\tau_2\} \rightarrow \{\tau_1 \circ \tau_2\} \\ & & \text{if } \tau_i \leq [], \mathcal{A}(\tau_1) \cap \mathcal{A}(\tau_2) = \emptyset \end{array}$$

where the function $\mathcal{A}$ returns the set of attributes of a relation.

3.2 Schema Instantiation of GPEs

The *S_inst* operator is applied on a set of tuples with the following parameters: a sequence of attribute and path variables and a type restriction. The operator extends the tuples contained in the set with all the possible instantiations of the path and attribute variables satisfying the type restriction. The operator signature is:

$$\begin{array}{rcl} S\_inst_{P_1 \ldots P_n;p} & : & \{\tau\} \rightarrow \{\tau \circ [P_1 : \tau_1, \ldots, P_n : \tau_n]\} \\ & & \text{if } \tau \leq [], \\ & & p : [P_1 : \tau_1, \ldots, P_n : \tau_n] \rightarrow Bool, \\ & & \mathcal{A}(\tau) \cap \{P_1, \ldots, P_n\} = \emptyset\} \end{array}$$

where the P_i's denote path or attribute variables. In the first case τ_i is the type $Path$, in the second the type Att. The operator definition is:

$$\begin{array}{rcl} S\_inst_{P_1 \ldots P_n, p(P_1, \ldots, P_n)}(e) & = & \{x \circ [P_1 : x_1, \ldots, P_n : x_n] \mid \\ & & x \in e, x_1 \in dom(P_1), \\ & & \ldots, x_n \in dom(P_n), \\ & & p(x_1, x_2, ..., x_n)\} \end{array}$$

where the P_i stands for either path or attribute variables (note that the order of the variables is irrelevant). The domain of an attribute variable is the set of all the attributes in the database schema. The domain of a path variable is the set of all the "legal paths" that can be constructed from the database schema. We are not really concerned about the interpretation given to "legal path". Ours is similar to that of [CACS94] but any reasonable interpretation would do. The restriction on the domains is given by the predicate p. Note that the input set is not involved in the instantiations of the path/attribute variables. This means, as we will see in the final section, that the operator can be applied on an empty set. Let us illustrate this operator by means of an example:
$S\_inst_{A, \alpha(A) \leq Person \wedge \omega(A) \in \{string,\ int\}}(\{[p : p1], [p : p2]\})$

The operator is applied on a set containing two tuples representing two persons whose identifiers are $p1$ and $p2$. The type restrictions require that the attribute variable A is applied to an object of type Person or subtype thereof and results in an integer or string (α denotes the start (codomain) of A, and ω the end (domain) of A.) This operation results in the following:

{ [p: p1; A: age]
[p: p1; A: name]
[p: p2; A: age]
[p: p2; A: name] }

Some Important Remarks on S_inst We represented attributes by their name. It is obvious that an attribute is more than a name. It captures type information, possibly an identifier or offset, etc. The same is true for paths. The issue of the representation of attribute/path is system dependent and will not be addressed in this paper.

The operation looks rather inefficient since it implies some sort of Cartesian product. However, (1) it is needed by the language and (2) when used as an intermediary operation it does not have to be evaluated as such. Before we clarify the second point, let us consider the following (rather silly) user query without any ω-type restriction.

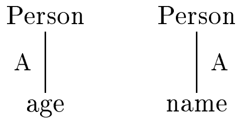
select *
from Person{p}.A

As we will see in the next section, *S_inst* operations are usually intermediate operators. From a physical point of view, it is obvious that, in the case of an intermediate operation, we can avoid unnecessary computation and redundancy. This can be done by considering *S_inst* as an operation annotating a set with all the possible instantiations of attribute/path variables. Hence, the annotated version should read:

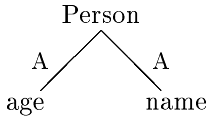
{ [p: p1;]
[p: p2;] $\}_{A \in \{age, name\}}$

Here, the representation of the information concerning the possible instantiations of A is preliminary. In fact, we will represent possible instantiations of attribute (or path) variables at the schema level as trees. Again, redundancy is the reason for this decision.

Consider again the attribute variable A with codomain *Person* and instances *age* and *name*. There exist two possibilities to represent its instantiations. Either, we use the two paths



or we use the tree



In this small example, we only save the duplication of *Person*, but when path variables are involved, the saving is tremendous. Further, since the semantics of path variables is restricted to acyclic instantiations at schema level [CACS94], trees are perfect for our representation. Indeed, this factorization is adequate for the paths instantiation through complex composition or inheritance graphs.

Note that the internal representation of paths is a physical issue. Neglecting the subtleties with empty sets, from an algebraic point of view we can view the result of the schema instantiation as a set of tuples. Nevertheless, the implementation of the algebraic operators work on the annotated relations and are adapted to them.

3.3 Data Instantiation of GPEs

The *D_inst* operator is applied on a set of tuples, some attributes of which are of the path or attribute sort. A subscript contains GPEs (that is an ordered sequence of path and attribute variables applied on an instance variable) that will be instantiated together with a restriction on these instantiations. The operator signature is:

$$D\_inst_{\{a_1:f_1,\ldots,a_n:f_n\};p}:$$

$$\begin{aligned}&\{\tau\} \rightarrow \{\tau \circ [a_1:\tau_1,\ldots,a_n:\tau_n]\}\\ &\text{if } \tau \leq [], f_i:\tau \rightarrow \tau_i,\\ &\mathcal{A}(\tau) \cap \{a_1,\ldots,a_n\} = \emptyset\\ &p:\tau \circ [a_1:\tau_1,\ldots,a_n:\tau_n] \rightarrow \ \mathit{Bool}\end{aligned}$$

Its definition is:

$$D\_inst_{\{a_1:V_1 P_{1_1}\ldots P_{n_1},\ldots,a_n:V_n P_{1_n}\ldots P_{n_n}\};p(a_1,\ldots,a_n)}(e) =$$

$$\begin{aligned}&\{x \circ [a_1:x_1,\ldots a_n:x_n] \mid\\ &\quad x \in e,\ x_1 = \mathit{apply}_{x.P_{1_1},\ldots,x.P_{n_1}}(x.V_1),\\ &\quad \ldots,\ x_n = \mathit{apply}_{x.P_{1_n},\ldots,x.P_{n_n}}(x.V_n),\\ &\quad p(x_1,\ldots,x_n)\}\end{aligned}$$

where the V_i's and P_i's are, respectively, instance and variable/path attributes. The function *apply* applies a path (given in a subscript) to an object or value.

Let us illustrate this operator with our previous example.

$D\_inst_{a:pA}$({ [p: p1; A: age]
[p: p1; A: name]
[p: p2; A: age]
[p: p2; A: name] })

The result is:

{ [p: p1; A: age, a: 35]
[p: p1; A: name, a: "John"]
[p: p2; A: age, a: 37]
[p: p2; A: name, a: "Mary"] }

Once again, some factorization may be performed so as to avoid unnecessary redundancy. Whereas schema instantiation could be captured by an annotated set, here we have to annotate tuples inside a set.

Also note that union types are required. This does not come as a surprise since GPEs usually introduce union types. To avoid unnecessary problems, we consider here marked unions as in [CACS94].

4 Optimizing Queries with GPEs

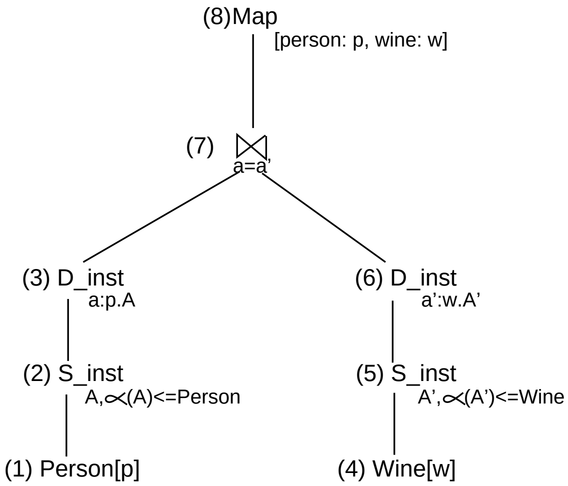
Traditional optimizers first perform type inference. After this step, all type information is present and the "real" optimization can start. Since the optimization decisions for queries containing generalized path expressions, often depend on the type information and the size of the subschema involved, type inference, query optimization and query execution must be interleaved. We explain in the next section how an optimizer can perform this interleaving of steps. In this section, we only demonstrate the necessity of interleaving by means of an example.

We concentrate on the demonstration of some optimization techniques. Some of these optimization techniques are simple extensions of existing techniques such as pushing selections, others are specific to queries with GPEs. These optimizations are essentially applications of algebraic equivalences. The two new operators *S_inst* and *D_inst* are reorderable with existing algebraic operators and with themselves, as long as they do not depend on each other in terms of the information consumer/producer relationship. This is very much like in standard relational or object algebra. Obviously, we cannot give examples for applications of all possible equivalences. Hence, we concentrate on some. In doing so, we pursue the goal of demonstrating that our approach is amenable to the optimization of queries with GPEs. The following section briefly indicates how the rewriting techniques can be integrated into an optimizer.

Simple Example The first example is that of the birthday present used in the previous sections. It will be used mainly to show how a query is translated into the algebra and how *D_inst*/*S_inst* operators are integrated in the rewriting process.

select struct(person:p,wine:w)
from Person{p}.A, Wine{w}.A'
where p.A = w.A'

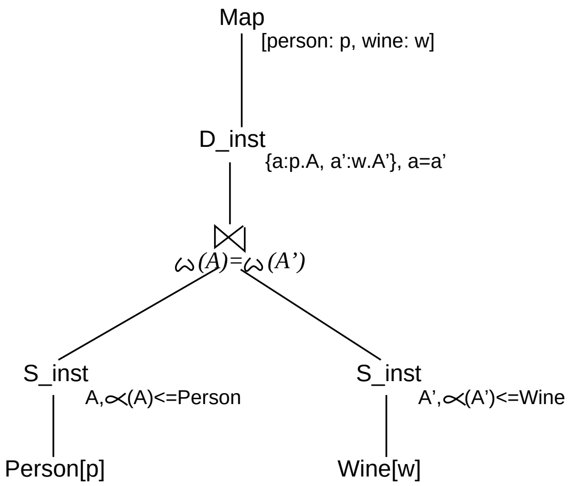
In a first step the query is translated into the algebra in the following way (for more details on the translation see [CM95a]).



Note that a static type inference is performed before/during the translation process and that the *S_inst* operations use the type information thus obtained. For instance, the restriction for the codomain of the attribute variable A is $\alpha(A) <= Person$ and that of A' is $\alpha(A') <= Wine$.

Operation (1) allows us to view persons as a set of tuples. Operation (2) finds all the possible attributes of a person. Operation (3) evaluates these attributes for each person. Operations (4,5,6) are similar. Operation (7) is a join. Two remarks are noteworthy: First, the join predicate involves union types (for more details on the manipulation of unions see [CACS94]). Second, the join operation has required some rewriting after the translation process. Usually, **from-where** clauses are translated into d-join and a selection. When the second parameter of the d-join is not dependent on the first, it can be rewritten into a Cartesian product or, using a selection, into a join. The last operation (8) is a map that builds the final result.

Sometimes it is useful to first perform a join on the type level and then use *D_inst* subsequently to instantiate the data. As we will see below this optimization is more useful for GEPs with attribute than path variables. This is the case in our example, where the end types of the attribute variables A and A' are equal ($\omega(A) = \omega(A')$) as can be inferred from the join predicate. Thus, the algebraic expression can be rewritten in the following way:



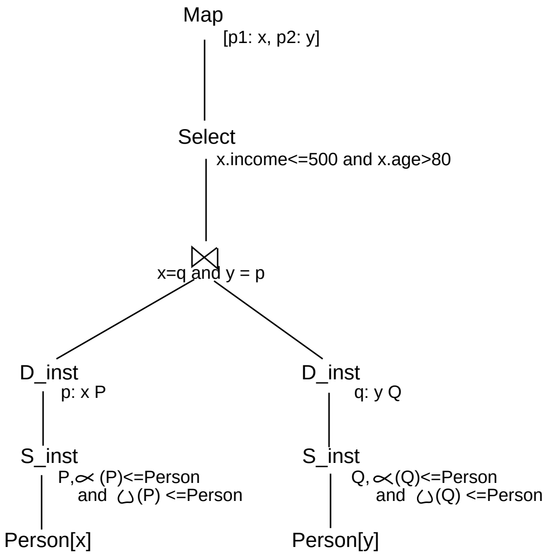
In the above example, the rewriting avoids the instantiation with all the attributes of Person and Wine. A simple evaluation of the above algebraic expression offers many similarities with the naive approach of the previous section. But instead of a number of unioned well-typed queries, the fact that we have here a number of attribute/path instances manipulated by the algebra, offers the possibility of considerably reducing the input to the optimizer and the number and size of intermediate results.

A More Complex Example We now consider a more complex example which will allow us to demonstrate more interesting rewriting techniques.

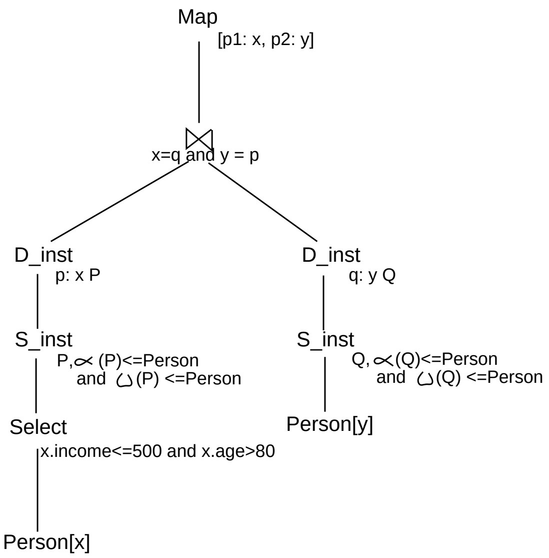
Assume we are interested in the social contacts of elderly persons having a small income. For this, we would issue the query

select struct (p1: x, p2: y)
from Person{x}P, Person{y}Q
where x.income $\leq$ 500 and
x.age > 80 and
xP = y and yQ = x

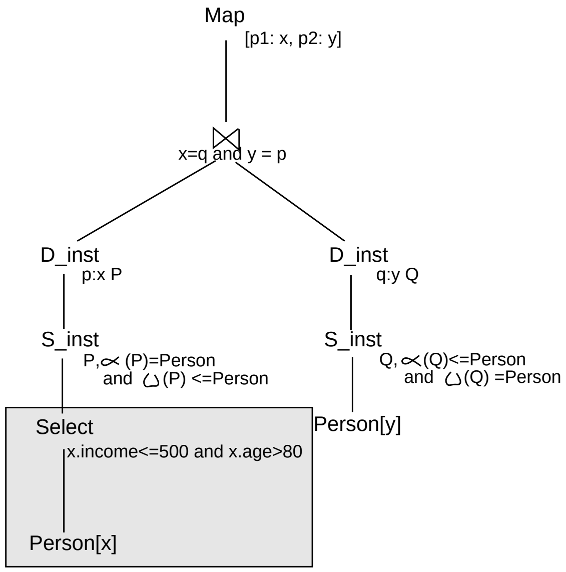
Assume that *Person* has many subtypes like *Employee, Professor, Pupil, Student, Manager, CEO, Staff, Secretary* and so on. Each of these subtypes is equipped with several attributes pointing to bosses, subsidiary managers, secretaries, project leaders, other project members, room mates, office mates, and so on. Hence, the possible instantiations of P and Q are immense. Note that factorization of these instantiations can save a little, but we do not elaborate on this kind of factorization in the current paper. Let us now translate the query into the algebra:



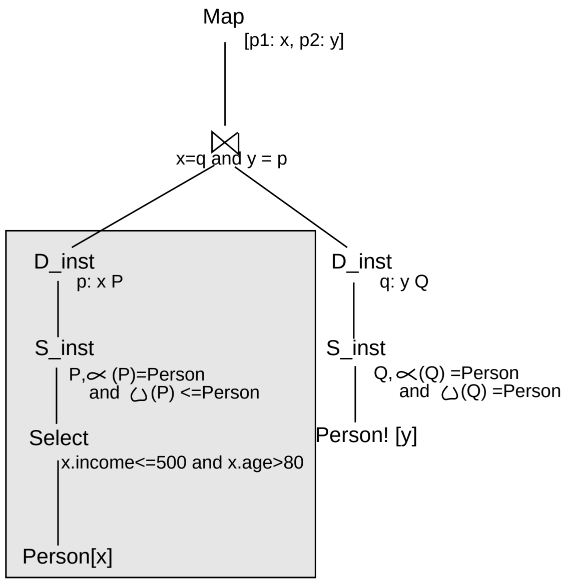
Note that we could statically get two indications concerning the type of path variables P and Q. They both start and end in an object of type Person or one of its subtypes. The ω-type information is obtained due to the condition in the join. We can now push the selection on *Person*s down:



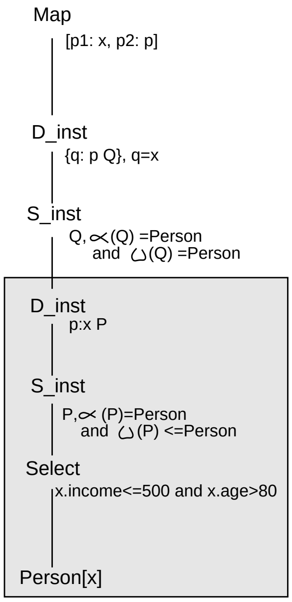
Now, a critical step is to come. The optimizer has to decide to make some partial evaluation first, before further optimization decisions are taken. The decision is to evaluate the selection on the set of persons. Then, not surprisingly, considering the selection, every qualifying object x has dynamic type *Person*. No object of any subtype of *Person* occurs. In this way the optimizer can replace $\alpha(P) \leq Person$ and $\omega(Q) \leq Person$ by $\alpha(P) = Person$ and $\omega(Q) = Person$ in the *S_inst* operations. The result of this optimization is illustrated below, where the evaluated expression is shaded. Note that this step of using the dynamic instead of the static type of x heavily restricts the search space for both *S_inst* operations.



The optimizer decides to proceed carefully and evaluate the first *S_inst* and *D_inst* operations. As it turns out, the persons x happen to have relations only with *Person*s; no instances of subtypes of persons occur at the end of the instantiated P. Consequently, the dynamic type of y can be restricted to *Person*, excluding its subtypes. Thus, we can restrict considerably the range of y by considering only the *Person*s that are not employees, etc. This can, of course, be achieved only if the system has a clever way of maintaining extents (e.g., as the union of all subextents). The result of this optimization is illustrated below, where *Person!* denotes that no subtype instances are scanned in the class Person.



We could be satisfied with this result, but obviously, the remaining scan over *Person!* is not necessary, since the values for y are already present at the end of the instantiated P. However, Q has to be evaluated. Using some appropriate rewriting rules, we end up with the following algebraic expression where the join has been eliminated.



Note that, if we are not interested in duplicates, the last *D_inst* operation just has to check the existence of a path Q such that $q = x$.

5 Extending an Optimizer

As demonstrated in the previous section, optimization of queries containing GPEs is very similar to traditional optimization of object queries: we rely on a set of equivalences which are applied by the optimizer in order to result in a (near) optimal plan. However, there are also some difficulties. By these, we are not referring to the two additional algebraic operators and the dozens of new equivalences. These can easily be dealt with by extensible optimizers [Bat87, HFLP89, GM91, GD87, KMP93, MZD92].

The key issue here is the partial evaluation of plans before further optimization decisions are made, or to be more precise, the evaluation of partial plans. In addition, another interesting issue is the use of indexes for evaluating *D_inst* operations. In this section, we briefly investigate these two issues.

5.1 Partial Evaluation

In the example of the previous section, we showed that the information resulting from a partial evaluation, i.e. statistical and type information, could be advantageously used for subsequent optimizer decisions. This raises two questions: (i) how can we interleave evaluation and optimization and (ii) when should we do it?

The how-question can be answered in a few words. Interleaving can be easily implemented into an interpreter. The interpreter has to evaluate queries partially and use the resulting type information to simplify the remaining expression. A compiler could be implemented such that it generates different alternative plans depending on the possible outcomes, and then generates an appropriate *chose-plan* operator [GW89] tying the different plans together.

The when-question is more complicated. Indeed, one cannot expect the optimizer to partially evaluate all alternative query execution plans just to perform further optimization. In a nutshell: partial evaluation is needed to do further optimization but should be performed with circumspection. Nonetheless, the problem sounds familiar even from the relational context: there, database statistics and cost functions were introduced in order to solve the problem. Further, within the context of object queries, the problem is faced when optimizing within a class hierarchy context [CM95b].

Our current approach to the problem is as follows. We rely on heuristics to push cheap operations, especially selections, after the *S_inst* and *D_inst* operators. More specifically, operators less expensive than *S_inst* or *D_inst* operators are pushed after them, the others are pulled before them. The building block approach seems to be well suited for supporting this task [KMP93, Loh88].

The remaining problem then is to estimate the costs of the *S_inst* and *D_inst* operators as well as information about the number of schema paths/path instances and classes/objects at the start and end of a path. Let us discuss the *D_inst* operator first since its treatment is easier. The *D_inst* operator starts with a set of schema paths to be evaluated, i.e. instantiated. To estimate the costs and number of instances touched, regular cost models developed for the evaluation of path expressions suffice [BF92, KM90].

For the *S_inst* operator, things are not so easy. To evaluate its cost and derive information about the number of classes at the beginning or end of a schema path, we rely on statistics that are gathered from the schema. For each class in the schema we keep four statistics: (1) the number of schema paths emanating from the class and (2) the number of classes at the end of these paths, (3) the number of schema paths ending in the class and (4) the number of classes at the beginning of these paths. These statistics are then used to estimate the costs and the cardinality of an *S_inst* operator. More specifically, we assume equal distribution of classes to schema paths. For example, if the starting class C is fixed and the path must end in a set S of classes, we estimate the number of schema paths by multiplying the total number of paths emanating from C by the factor resulting from dividing the cardinality of S by the number of classes in which the schema paths emanating from C end. We are well aware that for very large schemata, this approach seems to be too expensive. Hence, we expect future research on this issue.

5.2 Using Indexes on D_inst operations

There are several kinds of indexes [Ber89] (path indexes, multi-indexes, etc.) for object-oriented databases, some of which are implemented in real systems. Obviously, it is our goal to use indexes on the *D_inst* operations.

The use of standard object indexes offers some interesting challenges. First, the exact paths involved in a query are not known at the beginning of the optimization process. Again, we are back to interleaving of evaluation and optimization. Second, we are not dealing with queries along one or two paths but with queries along a great number of them. This implies clever splitting of *D_inst* operations. Similarly, whereas indexes usually concern paths of reasonable length, we are dealing with queries along potentially long paths. This can be solved by the use of standard path splitting techniques (e.g., [JWKL90]).

These difficulties and the fact that GPEs were first introduced for documents led us to look at full text indexing techniques. Obviously, these techniques are not reasonable in an environment with many updates. However, their interest is clear if we consider retrieval applications (e.g., interrogation of a library). The coupling of a database system with a full text indexing mechanism requires (i) a translation of the database into a document and (ii) the ability to retrace the path that leads from the root of a document to a particular string. The first point is easy, the second more tricky but can be done at a reasonable (storage) cost [Sim95]. Now, consider the following query:

select f
from Encyclopedia P(f)
where f.caption **like** "*Mont-St-Michel*"

The query retrieves the figures (or other elements featuring a caption attribute) of the encyclopedia whose caption contains the name "Mont-St-Michel". It is reasonable to imagine that there are at least a dozen schema paths leading to a caption attribute and many data (instantiated) paths. Hence, the need to use an index. Now, we have a full text index facility that can answer the query:

att:caption and Mont-St-Michel

Note that the query merges retrieval on attribute names and on contents. The result of this query is the intersection of the set of paths leading to objects featuring a caption attribute with the set of paths leading to objects containing the string "Mont-St-Michel". This could be:

$$\begin{aligned} \{\ & .chapters\{c_1\}.introduction\{i_1\}.figure(f1) \\ & .chapters\{c_1\}.articles\{a_2\}.figures\{f_2\} \\ & .chapters\{c_1\}.articles\{a_4\}.figures\{f_3\} \\ & .chapters\{c_2\}.articles\{a_1\}.references\{f_2\}\} \end{aligned}$$

where the c_i, a_i, f_i, i_i's are database object identifiers, "$\{\}$" represents the crossing of a set, "." the selection of an attribute and "()" sometimes gives the value of an attribute in a path. Note that we may have end-path objects containing the string "Mont-St-Michel" not in the value of their attribute caption but elsewhere (e.g. in the label). To use this (obviously interesting) index, we just have to rewrite the algebraic expression:

$$D\_inst_{\{f:EncyclopediaP\},f.caption \text{ like } ``*MontStMichel*"} \\ (S\_inst_{P,\alpha(P)\leq Book,\omega(P)\leq\{[caption:string]\}}\{\})$$

into something that could look like:

$$\sigma_{f.caption \text{ like } ``*Mont-St-Michel*"} \\ (FTI\_inst_{att:caption \text{ and } MontStMichel,P,f} \\ (Encyclopedia))$$

where FTI_inst is the operation invoking the full text index. The selection operation is needed to eliminate the objects containing the string "Mont-St-Michel" in attribute other than caption.

We believe that the interest of full text indexes is not limited to GPEs but can be used advantageously for all queries involving complex paths evaluation. In addition, appropriate schema indexing techniques can be used during the evaluation of the *S_inst* operation in order to instantiate attribute or path variables efficiently.

6 Conclusion

We have proposed an algebraic framework for the optimization of object query languages featuring generalized path expressions. In comparison with the approaches proposed so far and sketched in Section 2, our approach exhibits many advantages. By allowing GPEs to be captured by algebraic operations, it avoids the exponential input to the query optimizer, allows a compact representation of intermediate results and offers more flexibility in the ordering of operations.

Also, we have shown how an optimizer could be extended in order to incorporate the techniques we introduced and raised interesting issues concerning the use of full text indexing facilities in the database context.

Acknowledgements: The authors thank Yannis Ioannidis for fruitful discussions and Michel Scholl for many valuable comments on the final version of the paper.

References

[Bat87] D. Batory. Extensible cost models and query optimization in Genesis. *IEEE Database Engineering*, 10(4), November 1987.

[Ber89] E. Bertino. Issues in indexing techniques for object-oriented databases. In *Proc. of Advanced Database System Symposium*, pages 151–160, 1989.

[BF92] E. Bertino and P. Foscoli. An analytical cost model of object-oriented query costs. In *Proc. Persistent Object Systems*, pages 151–160, 1992.

[BMG93] J. Blakeley, W. McKenna, and G. Graefe. Experiences building the Open OODB query optimizer. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 287–295, 1993.

[BRG88] E. Bertino, F. Rabitti, and S. Gibbs. Query Processing in a Multimedia Document System. *ACM Transactions on Office Information Systems*, 6(1):1–41, January 1988.

[CACS94] V. Christophides, S. Abiteboul, S. Cluet, and M. Scholl. From structured documents to novel query facilities. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 313–324, Minneapolis, Minnesota, May 1994.

[CM95a] S. Cluet and G. Moerkotte. Classification and optimization of nested queries in object bases. Technical Report 95-6, RWTH Aachen, 1995.

[CM95b] S. Cluet and G. Moerkotte. Query optimization techniques exploiting class hierarchies. Technical Report 95-7, RWTH-Aachen, 1995.

[GD87] G. Graefe and D. DeWitt. The EXODUS optimizer generator. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 160–172, San Francisco, 1987.

[GM91] G. Graefe and W. McKenna. The Volcano optimizer generator. Tech. Report 563, University of Colorado, Boulder, 1991.

[GW89] G. Graefe and K. Ward. Dynamic query evaluation plans. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 358–366, 1989.

[HFLP89] L. M. Haas, J.C. Freytag, G.M. Lohman, and H. Pirahesh. Extensible query processing in Starburst. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 377–388, 1989.

[JWKL90] P. Jenq, D. Woelk, W. Kim, and W. Lee. Query processing in distributed ORION. In *Proc. Int. Conf. on Extended Database Technology (EDBT)*, pages 169–187, Venice, 1990.

[KKS92] M. Kifer, W. Kim, and Y. Sagiv. Querying object-oriented databases. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 393–402, 1992.

[KM90] A. Kemper and G. Moerkotte. Access support in object bases. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 364–374, 1990.

[KM93] A. Kemper and G. Moerkotte. Query optimization in object bases: Exploiting relational techniques. In *Proc. Dagstuhl Workshop on Query Optimization (J.-C. Freytag, D. Maier und G. Vossen (eds.))*. Morgan-Kaufman, 1993.

[KMP93] 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.

[Loh88] G. M. Lohman. Grammar-like functional rules for representing query optimization alternatives. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 18–27, 1988.

[MZD92] G. Mitchell, S. Zdonik, and U. Dayal. An architecture for query processing in persistent object stores. In *Proc. of the Hawaiian Conf. on Computer and System Sciences*, pages 787–798, 1992.

[QRS+95] D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, and J. Widom. Querying semistructured heterogeneous information. In *Proc. Int. Conf. on Deductive and Object-Oriented Databases (DOOD)*, pages 319–344, Dec. 1995.

[Sim95] J. Siméon. Recherche en texte intégral et bases de données orientées-objet. Master's thesis, Université de Nancy I, 1995.