Optimization of Object-Oriented Recursive Queries using Cost-Controlled Strategies *

Rosana S.G. Lanzelotte† Patrick Valduriez, Mohamed Zaït
Projet Rodin, INRIA, Rocquencourt, France

Abstract

Object-oriented data models are being extended with recursion to gain expressive power. This complicates the optimization problem which has to deal with recursive queries on complex objects. Because unary operations invoking methods or path expressions on objects may be costly to execute, traditional heuristics for optimizing recursive queries are no longer valid. In this paper we propose a cost-based optimization method which handles object-oriented recursive queries. In particular, it is able to delay the decision of pushing selective operations through recursion until the effect of such a transformation can be measured by a cost model. The approach integrates rewriting and increases the optimization opportunities for recursive queries on objects while allowing for efficient optimization.

1 Introduction

An advantage of object-oriented DBs (OODBs) is the direct modelling of complex objects. The value of an object attribute can be an object (possibly of the same class) or a collection of objects. Thus, a query in an OODB query language, such as ESQL [GV92] or O2Query [BCD89], may reference attributes of objects through path expressions [MS86], e.g., $O_1.A_1.A_2 \ldots A_n$ where O_1 is an object and each A_i an attribute of an object O_i referencing an object O_{i+1} or a collection of objects.

Most of the work in OODB query optimization has concentrated on optimizing path traversals, e.g., by exploiting path indices [MS86] or predicate selectivity [KM90]. In [LVZC91], we have proposed a comprehensive approach for optimizing non-recursive queries on objects, including path expressions, selects and joins, which allows any interleaving of operations. The optimizer actions are specified in a way that their impact on the cost of the execution plan is directly computable. Unlike most rewriting approaches (e.g., [KM90]), this approach enables the application of deterministic search strategies.

OODB data models are now being extended with recursion to gain expressive power [AK89, GV92]. For instance, object-oriented recursive queries are important in engineering DBs [CS90], e.g., execute a method for each subpart (recursively) connected to a given part object. However, no comprehensive attempt has been proposed to optimize recursive queries on objects. This is surprising given the large amount of work on recursive query processing in deductive DBs [BR86].

The reason may be the difficulty of keeping the typical dichotomy between *rewriting* and *optimization*. Rewriting is heuristics-based and proceeds by applying transformations to the initial query supposing they do not incur any loss of optimality. No selection among alternatives is involved because the heuristics are always considered worth applying. On the contrary, optimization proceeds by applying actions whose effect is measured by a cost function in order to select an optimal solution. The isolation between the two steps essentially reduces the complexity of optimization.

Recursive query processing in deductive DBs typically proceeds by rewriting using a simple heuristic: restricting the computation of recursive predicates to the *relevant* facts is better. Similar to pushing selection through join, this is achieved by pushing selection through recursion [BR86]. In OODBs, however, selections involving method calls or path traversals on complex objects may be expensive to compute. Therefore, pushing selection through recursion needs to be decided in the presence of a cost model. Thus, a task that is typically performed by the query rewriter becomes a matter of cost-based optimization. This observation suggests a different approach which integrates somehow rewriting and cost-

*This work was partially funded by the Esprit project EDS.

†Visiting INRIA, on leave from Pontificia Universidade Catolica do Rio de Janeiro (PUC-RIO).

1992 ACM SIGMOD - 6/92/CA, USA
© 1992 ACM 0-89791-522-4/92/0005/0256...$1.50

based optimization.

In this paper, we propose a cost-based approach to optimize recursive queries on objects. Unlike rewriting approaches where the transformations apply to the conceptual query, we propose an optimizer that acts upon physical entities. Thus, the impact of the optimizer actions on the cost of an execution plan is directly computable. In particular, our approach delays the pushing of selective operations (i.e., selections and joins) through recursion until the cost of a plan can be estimated. In order to better integrate rewriting and cost-based optimization tasks (e.g., join enumeration) and to reduce the overall complexity, the optimizer is able to focus on *subproblems* of different granularities, from a path or a select-project-join (spj) to the entire query. As the optimizer actions are controlled by cost-based strategies, the optimizer is able to investigate more general solutions than in previous work, while guaranteeing optimality in an acceptable time.

The paper is organized as follows. Section 2 introduces a model for object-oriented recursive queries as a support for optimization. Section 3 introduces a model for query execution plans and an associated cost model which considers several options for storing complex objects (e.g., multiclass clustering, decomposition, path indices). In Section 4, we propose an optimization approach that integrates rewriting and cost-based optimization. Also, we show that this approach is general enough to implement several optimization techniques. Section 5 concludes.

2 A Model for Recursive Queries on Objects

In this section we propose a model for representing object-oriented recursive queries. The main goal is to provide a canonical input representation for the optimizer which support the capabilities of OODB query languages. Our query model is derived from the System Graphs [KL86] and enables us to express queries involving path expressions, joins, recursion and method calls. By easing the factorization of overlapping path expressions, it is well-suited for optimization.

2.1 A Conceptual DB Schema

The conceptual model deals with *classes*, whose instances are *objects*, and *relations*, whose instances are *values*. There is a mapping from each class or relation name to a type. Types are built from atomic types and the usual tuple, list and set constructors (denoted [ ], <> or { } respectively). An *instance* of the conceptual schema associates *extensions* (i.e., the set of instances) to class and relation names of the schema. The conceptual schema of Figure 1 will serve as a basis for the sample queries of the paper.

```
class Person:                    class Composition:
[ name: string                   [ title: string,
  birth: date                      author: Composer inverse
  age: integer has                     of Composer.works,
  value computeAge() ]             instruments: {Instrument} ]

class Instrument:                class Composer: isa Person and
[ name: string,                  [ master: Composer,
  family: string ]                 works: {Composition} ]

relation Play: [ who: Person, instrument: Instrument ]
```

Figure 1: A Sample Conceptual Schema

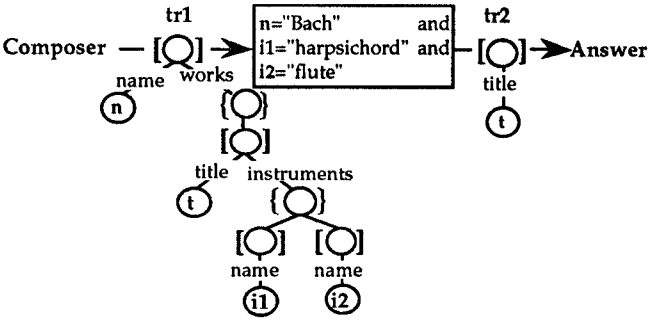

Figure 2: A Query Graph

We say that Composer is a *subclass* of Person. The model allows for specifying that an attribute is the inverse of another one (e.g., Composition.author is the inverse of Composer.works). Methods are considered as *computed* attributes (e.g., *age* in class Person).

2.2 Query Graphs

Queries are represented by means of *query graphs*. Figure 2 shows the query graph for a query that retrieves "the title of the works of Bach including a harpsichord and a flute".

A query graph includes *predicate nodes*, represented by squares, *name nodes* (e.g., **Composer** and **Answer** in Figure 2), and directed arcs connecting name nodes to predicate nodes and vice-versa. Name nodes correspond to relation or class names of the conceptual schema. A predicate node has one or more *incoming arcs* originated at name nodes, one Boolean predicate and one *outgoing arc*. The name node that is the destination of an outoing arc is called the *output name node* of the predicate node (e.g., **Answer** in Figure 2). The *incoming arcs* are labelled by trees (e.g., **tr1**) which indicate, by means of variables (e.g., **n**, **t**, **i1**, **i2**), the subobjects needed in the predicate or in the outgoing arc of a predicate

node[1]. The *outgoing arcs* are labelled by trees (e.g., **tr2**) which indicate the type of the predicate node output. To denote the output projection, they reference the variables in the trees on the incoming arcs (e.g., **t**). The structure of a *tree label* is derived from a type of the conceptual schema (e.g., the structure of **tr1** is derived from the type of Composer). In the tree labels, atomic typed nodes are denoted by a circle and non-atomic typed nodes are denoted by circles surrounded by the corresponding type constructor (i.e., { }, [ ] or <>).

The semantics of a query graph is as follows. Each predicate node denotes an spj applied to the instances of the incoming name nodes (e.g., **Composer** in Figure 2); the order in which the operations are performed is not fixed. The result is stored in the output name node (e.g., **Answer** in Figure 2).

We denote a query graph by the set $\mathbf{Q} = \{(Name \leftarrow p)_i\}$, i≥1, where each p is one of its predicate nodes and $Name$ is the output name node of p. A predicate node p is denoted as $SPJ(In, pred, outproj)$, where In is its set of incoming arcs, *pred* is the Boolean predicate and *outproj* is the tree label of its outgoing arc. An incoming arc is denoted as a pair $(Name, tree)$, in which $Name$ is its name node and *tree* is its tree label. Thus, the query graph of Figure 2 is denoted as

Q= {(**Answer** ← SPJ({(**Composer**,**tr1**)}, (**n**="Bach" and **i1**="harpsichord" and **i2**="flute"), **tr2**))}

We denote a tree label or any subtree by a set $\{(Att, tree, variable)\}$ of its children. *Att* is NIL when a subtree does not implement a named attribute, e.g., a subtree corresponding to a set- or list-typed node. *Variable* is NIL if there is no variable associated with the node. A subtree corresponding to an atomic attribute is denoted by the empty set. For example, in Figure 2, **tr1** is {(name,trName,**n**),(works,trWorks,NIL)}, trName is { }, trWorks is {(NIL,trComposition,NIL)}.

2.3 Recursive Queries

Our query graphs are designed for expressing object-oriented recursive queries. Suppose that the conceptual schema of Figure 1 is extended with a recursive view Influencer defined as:

```
relation Influencer
includes (select [master:x.master, disciple:x, gen:1]
          from x in Composer)
union    (select [master: i.master, disciple:x,
                  gen:add1gen(i.gen)]
          from i in Influencer, x in Composer
          where i.disciple = x.master)
```

[1] These trees can be viewed as tree-shaped *adornments* [BR86] that depict the bindings of the input objects. In the relational model, adornments are strings (because relations are unidimensional) but in an object-oriented model they are trees.

Figure 3 shows a query that retrieves "the names of the composers influenced by composers for harpsichord that lived 6 generations before". Predicate nodes P1 and P2 define the recursive Influencer view, which is the "transitive closure" of Composer with respect to the *master* attribute. The instances of Influencer are the union of the output instances of predicate nodes P1 and P2. Predicate node P3 represents the query on the view. Note the similarity between query graphs and representations proposed for queries on sets of rules, such as System Graphs [KL86].

With a query like the one in Figure 3, most deductive query processors would push selection and projection through recursion [BR86]. The objective is to restrict the recursive computation to only the *relevant* facts needed for answering the query. Pushing the projection on the *name* attribute of the disciple does not incur any overhead, as it is an atomic attribute and can be retrieved when accessing the *master* attribute of a Composer. However, pushing the selection *i="harpsichord"* introduces a complex path expression, the path *master.works.instruments.name*, inside the recursion. If the query is rewritten at the conceptual level, it is not possible to tell which form is better. When objects and recursion are involved, such transformation cannot be blindly applied as a heuristic and must be decided in the presence of a cost model.

3 A Model for Query Execution Plans

The input to the optimizer is a query graph and its output a query *execution plan*. Here we are concerned with cost-based optimization, where the effects of the optimizer actions are measured by a cost function. To allow this, an execution plan must be expressed as a sequence of operations on *physical entities* unlike query graphs which deal with conceptual entities. This section presents our model for query execution plans (based on a physical schema) and the associated cost model.

We assume that the physical DB model follows the direct storage approach, in which the identifiers (oid's) of sub-objects are stored within the owner objects [VKC86]. This is the most frequent approach implemented in OODBs. This physical model allows for *clustering* the instances of the sub-objects close to the owner object record (e.g., in a same or neighbor disk page). A static clustering strategy is assumed, which is appropriate for general purpose transactions accessing several objects. The physical model also allows for *decomposing* extensions into horizontal or vertical fragments to optimize the processing of selections and projections. *Path indices* [MS86] are available for accelerating accesses that span over a whole hierarchy of nested attributes. A path index is denoted by the sequence of attributes which it spans. For example, a path index on

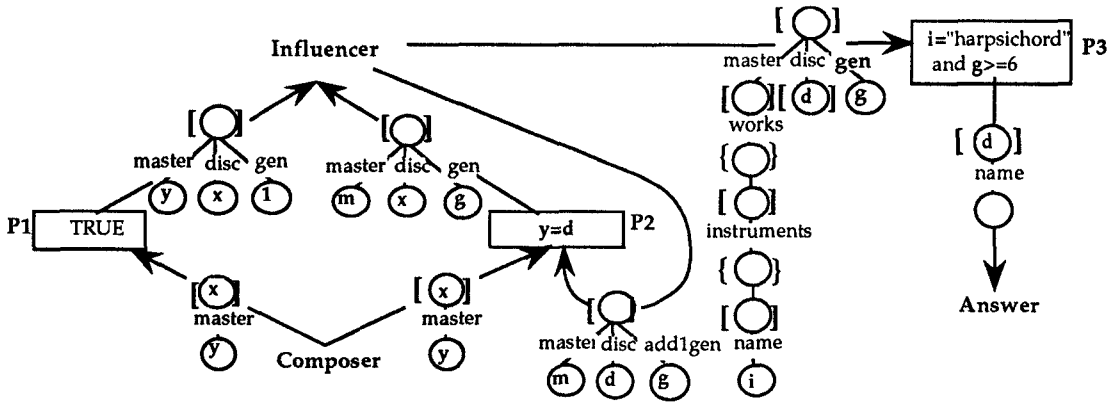

Figure 3: A Recursive Query

works.instruments is available, whose entries are triples composed of the oid's of the classes Composer, Composition, Instrument. This index speeds the access of the instruments used in the works of a Composer.

We say that an *atomic entity of the physical schema* is either a non-decomposed extension or a fragment of a decomposed extension. We say that two atomic entities are connected through an *implicit join* if they correspond to class extensions which implement an object and its sub-objects (e.g., the extensions of classes Composer and Composition are connected through the *works* attribute). The usual joins are called hereafter *explicit joins* (e.g., **y=d** in the query of Figure 3).

3.1 Execution Plans as Processing Trees

We model execution plans by means of *processing trees* (PTs) extending the definition of [KZ88]. Figure 4 shows two PTs for the query of Figure 3.

PTs can be considered as an algebra for specifying the query execution: the interior nodes are operators (e.g., join, union) and the leaf nodes are atomic entities of the physical schema referenced in the query.

Definition: A PT rooted at node N, denoted by $N(child_0, child_1, \ldots, child_{k-1})$, is such that each node N or $child_i$, $0 \leq i \leq k-1$, is either

- a projection **Proj**, a selection $\mathbf{Sel}_{pred}$ (k=1),
- an implicit join $\mathbf{IJ}_{attrName}$, an explicit join $\mathbf{EJ}_{pred}$, a fixpoint **Fix**, a union **Union** (k=2),
- an implicit join implemented by a path index $\mathbf{PIJ}_{pathIndex}$ (k≥2),
- an atomic entity of the physical schema or a temporary file (k=0).

We define PTs in such a way that they can be treated as functional terms. Thus, some of the optimizer actions can be specified as transformation rules applied to PTs. For example, the functional representation for the PT of Figure 4.(i) is **Answer** = $\mathbf{IJ}_{disc}(\mathbf{Sel}_{name="harpsichord"}(\ldots)$,Composer).

PT nodes also capture the specific algorithm used to implement an operation (e.g., in the case of **EJ** or **Fix** nodes), the materialization or pipelining of its output and the output projection. However, for clarity, we abstract these in the notation for PT nodes. Fixpoint is considered here as a paradigm for recursive queries. The *temporary* files mentioned in the case k=0 of the definition refer to the files used for storing intermediate results (e.g., Influencer). The *attrName* on an $\mathbf{IJ}_{attrName}$ node indicates the access to the attribute *attrName* of its left argument. An explicit predicate *pred* on a $\mathbf{EJ}_{pred}$ or $\mathbf{Sel}_{pred}$ node is a Boolean expression on paths, e.g., *name="harpsichord"*. A *pathIndex* indication on a $\mathbf{PIJ}_{pathIndex}$ node is a path to which corresponds an existing path index in the physical DB schema (e.g., *works.instruments*). These definitions rely on the uniqueness of attribute name! s, which can be enforced by renaming.

A PT models a bottom-up execution, where each node consumes operands from left to right. The execution ends at the root node, which corresponds to the distinguished name of the query graph (i.e., the query answer). The PT of Figure 4.(i) specifies the selection *i="harpsichord"* and the projection on the disciple *name* to be performed after the fixpoint. In Figure 4.(ii), both the selection and the projection have been pushed through recursion (applying the technique of [KL86]). Thus, the temporary file Influencer' already captures the selection *i="harpsichord"* and the projection on the *name* attribute of the disciple. The remaining selection *gen* ≥ 6 cannot be pushed and remains to be verified after the recursion in both PTs of Figures 4.(i) and 4.(ii). Note that pushing

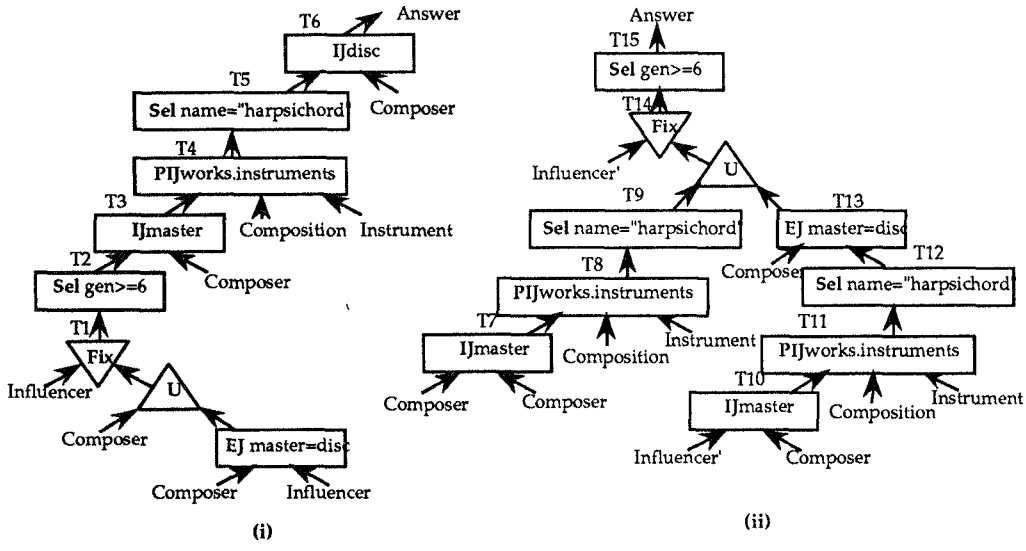

Figure 4: Processing Trees for the query of Figure 3

the projection does not imply any overhead: the attribute *name* of Influencer.disciple is obtained from Composer.name without any additional implicit join. This is why the implicit join on *disc* has disappeared in the PT of Figure 4.(ii). However, pushing the selection puts additional implicit joins inside the computation of the fixpoint! In deductive DBs, the PT of Figure 4.(ii) would be generated although it is probably not optimal. In order to select the optimal PT, a cost model is needed.

3.2 Cost Model

We now sketch a cost model that associates a cost estimate to each PT node, where cost refers to execution time. In a first step, we express the operation captured by a PT node in terms of the basic operations involved. In a second step, we derive cost formulas for PT nodes by combining the cost formulas for the involved basic operations. We generalize this process to compute the cost of an entire PT. Examples of basic operations are accessing an atomic entity of the physical schema and evaluating a predicate.

In the following, C_i refers to the name of an atomic entity of the physical schema (relation or class), N to a PT node, A_i to an attribute of the class C_i, and P to a predicate. We give the cost formulas for the basic operations.

- *access_cost*(C_i)[2]: the cost of accessing the instances of C_i. It depends on how the instances are stored and on whether an index is defined on C_i or not;
- *access_cost*(C_i, P): the cost of accessing the instances of C_i in the presence of predicate P. This access is performed using an index, if P references an indexed attribute of C_i, or is a sequential scan;
- *access_cost*(C_i, C_{i+1}): the cost of accessing the C_{i+1} instances referenced by one C_i instance through attribute A_i. The way this operation is performed depends on whether the C_{i+1} instances are clustered or not with the C_i instance that references them (directly or not);
- *eval_cost*(C_i, P): cost of evaluating the predicate P on all records stored in one page of C_i;

We assume a centralized cost model and that atomic entities are initially stored on disk. The computed cost includes I/O time and CPU time, thereby giving a fair estimation of the use of machine resources. We use the path index definition of [MS86] which we see as a generalization of join indices [Va87]. Selection or path indices are assumed to be implemented as B^+-trees.

We use the following parameters obtained from the physical schema description:

$\mid C_i \mid$: number of pages on which C_i is stored
$\parallel C_i \parallel$: number of instances of C_i
nblevels(I): number of levels of index I
nbleaves(I): number of leaves of index I

We also use the following functions:

- *nbpages*(C_i, P): returns the number of accessed pages, i.e., $\mid C_i \mid$ reduced by the selectivity of P;

[2] When estimating *access_cost*, we take into account the fact that some of the needed data are already in main memory and need not to be fetched from disk

PT node	cost formula
$Sel_{selpred}(C)$	$access\_cost(C, selpred) + nbpages(C, selpred) * eval\_cost(C, selpred)$
$EJ_{pred}(C_i, C_j)$	$access\_cost(C_i, pred) + nbtuples(C_i, pred) * (access\_cost(C_j, pred) + nbpages(C_j, pred) * eval\_cost(C_j, pred))$ [a]
$IJ_{A_t}(C_i, C_j)$	$access\_cost(C_i) + \| C_i \| * access\_cost(C_i, C_j)$
$PIJ_{pathInd}(C, C_2, \ldots, C_n)$	$\|\|C\|\| * (nblevels(pathInd) + nbleaves(pathInd)/\|\|C_1\|\|)$ [b]
$Fix(T, P)$	$\sum_{i=1}^{n} cost(Exp(T_i))$ [c]

[a] This formula is valid if the EJ operation is implemented using a Nested_Join Loop or Index_Join algorithm.
[b] The path index pathInd is defined on the path $C_1.A_1.\ldots.A_{n-1}$, where each A_i is an attribute of type C_{i+1} defined in class C_i.
[c] where n is the number of iteration in the loop of the semi-naive algorithm, $Exp(T_i)$ denotes the fixpoint equation Exp (contained in P), having T_i as input instead of T, and T_i denotes new tuples produced at step $i-1$.

Figure 5: Cost Formulas

- $nbtuples(C_i, P)$: returns the number of accessed tuples, i.e., $\| C_i \|$ reduced by the selectivity of P.

Figure 5 shows the cost of the PT nodes introduced in Section 3.1. For simplicity, we do not consider the cost of materializing the output of a PT node.

The cost of a PT rooted at node N, denoted $N(child_0, child_1, \cdots, child_{k-1})$,is computed as follows

$$cost(PT) = cost(N) + \sum_{i=0}^{k-1} cost(child_i)$$

4 Optimizing Query Graphs

In this section, we propose a cost-based optimization approach for object-oriented recursive queries, which allows to separately optimize *subproblems* (e.g., one path or one spj) to reduce the global complexity. Thus, the optimization granule, denoting a subproblem, may *vary* during optimization. Optimization proceeds with the following successive steps:

- **rewrite**: fixpoint recursion is identified. The **Union** and **Fix** operators, that were not explicit in the query graphs, are generated. The optimization granule is the entire query graph;
- **translate**: the query graph is translated onto the physical schema. Conceptual entities are replaced by physical entities and paths are converted into sequences of **IJ** and **PIJ** nodes. The granule is one arc of the query graph and its involved paths;
- **generatePT**: predicate nodes are optimized, which corresponds to optimizing spj's. **EJ** and **Sel** nodes are generated. The optimization granule is one predicate node. A PT (e.g., like the one in Figure 4.(i)) is generated for the query graph;
- **transformPT**: finally, the position of selective operators with respect to recursion is decided. The granule is again the entire query, but now in the form of a PT (generated by the previous steps). Reoptimization is performed when necessary.

Unlike most optimizers, the position of selective operators (i.e., selection and join) with respect to recursion is decided after a PT has been generated. As PTs have an associated cost estimate, the optimizer is able to measure the impact of such transformations.

4.1 Optimization Approach

We argued that the typical solution in deductive DBs (i.e., restricting recursion to the relevant facts) is not appropriate for object-oriented recursive queries. Another proposal consists of exhaustively enumerating all the solutions, each assigned a cost, in order to choose the least costly as optimal [KZ88]. As this strategy is cost-based, optimality is guaranteed, but the optimization time may become unacceptably high. Furthermore, this approach does not take advantage of the existence of *subproblems* that could be separately optimized (e.g., one path expression or one spj).

The optimization approach herein adopted generalizes the extensible one we proposed in [LV91], in which extensibility is achieved by isolating the specifications of the optimizer *search space* from its *search strategy*. The search space is characterized by the optimizer *actions* and their scope of application. The search strategy is responsible for controlling the application of such actions. The optimizer combines two paradigms: *procedural* for specifying search strategies and *declarative* for specifying transformation actions. Optimization is performed by the procedure **optimize** below:

```
optimize(Q)
{ rewrite(Q);
  for each (N,tree) of Q translate(N,tree);
  for each SPJ(In,pred,out) of Q | (∀N ∈ In) isaPT(N)
      Q := Q - {N←SPJ(In,pred,out)} ∪
```

{N←**generatePT**(SPJ(In,pred,out)) };
repeat **transformPT**(Q) until saturation; }

This strategy postpones pushing selective operations through recursion (i.e., performed by **transformPT**) after the generation of a solution PT for the query (i.e., performed by **generatePT**). This **optimize** strategy is one possible choice. Our approach is extensible and allows for specifying other optimization strategies. The condition (∀N ∈ In) isaPT(N) requires that all inputs N to the predicate node must have been previously optimized, thus forcing the optimization of the query graph to proceed bottom-up to enable cost computations.

Figure 6 summarizes the features of the procedures referred to in **optimize** with respect to their scope, strategy and types of PT nodes generated.

A procedural paradigm for specifying strategies fits well their procedural nature. But some optimizer actions, referred to and controlled by the strategies, are declaratively specified through transformation **actions** as in transformation-based optimizers. Transformation **actions** are suited to recognizing and transforming "patterns" occurring in their scope of application. They have the form:

action: **F** | **constraint** → **G**

where **action** is the action label, **F** and **G** are patterns describing subparts of the granule to which **action** is applied and **constraint** is a predicate whose truth conditions the applicability of the action. When **action** is applied to some object **O** (denoted **action(O)**), if **F** matches some part of **O**, and **constraint** is true, then the part matched by **F** in **O** is replaced by **G** (see the next sections for examples of transformation actions).

With these two paradigms, procedural and declarative, we are able to implement several optimization techniques and search strategies. In the rest of this section, we illustrate their use by specifying rewriting and optimization actions for query graphs.

4.2 Rewriting the Query Graph

In the present context, the purpose of rewriting is to recognize fixpoint recursion and to generate **Fix** and **Union** nodes that are not explicit in the query graphs. Rewriting is performed by the procedure **rewrite** below:

rewrite(Q) { repeat **union**(Q) until saturation;
repeat **fixpoint**(Q) until saturation; }

The above strategy is irrevocable: the **union** and **fixpoint** actions are applied up-to-saturation without any choices involved.

The purpose of the **union** action is to make the union operator explicit in query graphs.

union: **Q** | (Name ← p1) ∈ **Q** ∧ (Name ← p2) ∈ **Q**
→ **Q** − {(Name ← p1), (Name ← p2)} ∪
{(Name ← **Union**(p1, p2))}

The purpose of the **fixpoint** action is to recognize fixpoint recursion and to add **Fix** operators:

fixpoint: **Name** | (**Name** ← p) ∈ **Q**
∧ **fixpointRecursion(Name)** → **Fix** (**Name**, p)

The constraint **fixpointRecursion(Name)** verifies the ability of computing the recursion as the fixpoint of an equation referencing **Name**. In the query graph, this equation is captured by the predicate node p whose outgoing arc leads to **Name**. Other rewriting actions could be devised, e.g., for *folding* predicate nodes to eliminate non-recursive view definitions.

After rewriting, each **p** such that (**Name** ← **p**) ∈ **Q** may denote an **SPJ**, a **Fix** or a **Union** node (contrary to the original query graph, in which only **SPJ** terms occurred).

4.3 Translation to the Physical Schema

Cost-based optimization requires the query graph to be translated onto the physical schema. The main goal is to depict the atomic entities of the physical schema referred to in the query and the connections (i.e., due to implicit joins) between them. Each arc (N, tree) of the query graph is translated onto one sequence of **IJ** nodes[3] that implements it:

translateArc: (N,tr) | type(N)=[..., *Att*:C, ...] ∧
(*Att*,tr′,var) ∈ tr ∧ isaClass (C) → (**IJ**$_{Att}$(N,C),tr′)

Recall that a tree label tr is denoted by a set of triples (*Att*,tr′,var). For each attribute *Att* of N that is implemented by a class C (i.e., the type of N is [..., *Att*:C, ...]), the arc (N,tr) is replaced by an implicit join node **IJ**$_{Att}$ between N and C through the *Att* attribute. As **translateArc** is applied up to saturation, it stops the translation of one arc when the tree-labels of all arcs do not involve any more class extensions. This process may generate different sequences of **IJ**, as the tree-labels can be scanned in many ways. Analogous translation actions deal with the case where *Att* has type {C} or <C>.

Action **collapse** collapses subsequent **IJ** nodes into a **PIJ** (path implicit join) node, provided that there is an applicable path index. For example, it transforms the sequence of implicit joins **IJ**$_{instruments}$(**IJ**$_{works}$(**IJ**$_{master}$(Influencer, Composer), Composition), Instrument) into **PIJ**$_{works\ instruments}$

[3] There may be several such sequences and the choice among them is cost-based.

Procedure	Granularity	Strategy	PT nodes generated
rewrite	the entire query (graph)	irrevocable[a]	**Fix, Union**
translate	one arc	cost-based	**IJ, PIJ**
generatePT	one predicate node	cost-based (generative)	**EJ, Sel**
transformPT	the entire query (PT)	cost-based (transformational)	none

[a]An *irrevocable* strategy does not involve choices and proceeds always straight-ahead, like in query rewriters.

Figure 6: Summary of Optimization Steps

($\mathbf{IJ}_{master}$(Influencer, Composer), Composition, Instrument) that occurs in Figure 4. This is made possible by the existence of the path index on *works.instruments*.

collapse: $\mathbf{IJ}_{p1}(\mathbf{IJ}_{p2}(N1,N2),N3)$ | existPathIndex($p2.p1$) $\rightarrow$ $\mathbf{PIJ}_{p2.p1}(N1,N2,N3)$

All the atomic entities of the physical schema referred to in the query are present in the translated query graph. They are replicated for each occurrence, so that each ocurrence is the input of at most one predicate or **IJ** node. Thus, the translated query graph becomes a tree where the leaf nodes are atomic entities of the physical schema. Recall that we do not represent the output projection of each PT node for clarity, but each arc in the translated query graph is still labelled by simpler trees than the original ones. These labels are of depth one or two (i.e., in the case of the set- or list-valued nodes). Hereafter, we refer to an arc (N,tree) by N only and consider that the needed projections (which do not involve any more implicit joins) are also captured.

4.4 Generating PTs for Predicate Nodes

As in relational optimizers, the main goal when optimizing predicate nodes (i.e., an spj) is to build an optimal join permutation and the implementations for the involved operations (e.g., access methods to individual entities or join algorithms). As a consequence of **generatePT** (not shown here for space reasons), each predicate node SPJ(In,pred,out) is replaced in the query graph by the optimal PT that captures the chosen permutation. At this step, **EJ** and **Sel** nodes are generated. A *generative* strategy builds several PTs from the atomic entities of the physical schema [Se79], in a bottom-up fashion. The generated PTs are, then, compared with respect to their costs, in order to keep the least costly. In [LV91] we model several such strategies. They proceed by applying the actions **Sel** and **join** until saturation, i.e., the Boolean predicate of the predicate node has been completely consumed.

Action **sel**(**N**,pred) (resp. **join**) systematically *expands* N by adding **Sel** nodes (resp. **EJ**). At the same time, the Boolean predicate pred is "consumed".

sel: (**N**,**pred**) | **pred** = and(selpred(**N**),**pred'**) $\rightarrow$ ($\mathbf{Sel}_{selpred}$(**N**),**pred'**)

In the constraint part of the **join** action, the predicate disjoint(**N**,**Inner**) is true if the atomic entities captured by **N** and **Inner** are disjoint with respect to In (i.e., the set of inputs to the SPJ term). The requirement of the existence of a join predicate avoids the generation of PTs where there are Cartesian products.

join: (**N**,**pred**) | **Inner** $\in$ In $\wedge$ disjoint(**N**,**Inner**) $\wedge$ **pred** = and(joinpred(**N**,**Inner**),**pred'**) $\rightarrow$ ($\mathbf{EJ}_{joinpred}$(**N**,**Inner**),**pred'**)

As action **sel** is applied before **join**, **Sel** nodes are generated as soon as possible, according to the relational heuristics of pushing selection through join. After optimizing the predicate nodes by **generatePT**, the query graph resembles that of Figure 4.(i). All the predicate nodes were optimized and replaced by "locally" optimal PTs with an associated cost.

4.5 Transforming PTs: pushing selective operations through recursion

Recall that in Section 4.1, we proposed a control strategy that postpones **transformPT** to **generatePT**. This is analogous to two-pass search strategies [IC90]. The strategy of **transformPT** below starts by pushing selective operations, such as selection or join, through recursion using the **filter** action. Then, it tries to further improve the obtained PT through a randomized strategy, thus transforming the PT in Figure 4.(i) into that in Figure 4.(ii).

transformPT(PT)
{ newPT := PT;
filter(newPT);
newPT := **randOptimize**(newPT);
if cost(newPT) < cost(PT) then PT := newPT; }

Action **filter** pushes selection through recursion following the algorithm of [KL86]. A similar action is available to push join through recursion.

filter: $\mathbf{Sel}_{pred}$(**pt**(**Fix**(Rec,**Union**(Base,**pt'**(Rec))))) | canPush($pred$,Rec) $\rightarrow$ **Fix**(Rec,**Union**($\mathbf{Sel}_{pred}$(**pt**(Base)),**pt'**($\mathbf{Sel}_{pred}$(**pt**(Rec)))))

We assume that **pt**(X) or **pt'**(X) match any PT containing X as a subtree, i.e., they capture functional expressions containing X. In deductive DBs, it is typically assumed that there are no operations between the selection and the fixpoint (e.g., the **filter** rule would start as **Sel(Fix** ...)). In the present context, implicit joins may come between the selection and the fixpoint and the rule must be more general. The constraint *canPush* enforces the requirements for pushing the selection or join [KL86].

As some optimization alternatives (e.g., the choice of an access method) are based on specific bindings, shifting a portion of a PT may require its reoptimization. Thus, the strategy for **transformPT** applies a *randomized* strategy (e.g., Iterative Improvement [IC90]) to the filtered PT, which tries to transform it in order to further reduce the cost. The termination of a randomized strategy is conditioned by the optimization time or the stability of the current solution, meaning that it is unlikely that this solution can be further improved. Randomized strategies are convenient for reoptimization, because they can be customized to apply "convenient" transformations, once a portion of the PT has been shifted (e.g., use an applicable index). Also, they stop if the filtered PT is a stable solution that cannot be improved.

Our cost-based approach enables us to investigate solutions where *join* is pushed through recursion, not proposed before. A join may be very selective, making it worth to push it through recursion. Only in the presence of a cost model, one is able to judge of the convenience of such a transformation. For example, suppose a query that "retrieves the composers that were influenced by the masters of Bach". It is answered by a join between Influencer and Composer on the master attribute, i.e., Influencer.master = Composer.master and Composer.name = "Bach". It is clear that this join is very selective and, if pushed through recursion, would restrict the recursive computation of Influencer to few "relevant" facts.

4.6 A Comprehensive Example

To illustrate our optimization approach, we apply it to the query of Figure 3. First, the query graph is rewritten: the **Union** operator is generated whose destination is Influencer; the name node Influencer is replaced by **Fix**(Influencer,**Union**(...)) due to the detection of fixpoint recursion. The arcs of the query graph are translated onto the physical schema: all the conceptual name nodes are replaced by the atomic entities of the physical schema that implement the corresponding extensions; paths spanning several class extensions are translated onto sequences of **IJ** nodes that implement them. As the atomic entities are replicated for each occurrence, the translated query graph becomes a tree.

PT node	PT node cost
T_1	$\lvert Cpr \rvert * pr + \lVert Cpr \rVert * \lvert Cpr \rvert * (pr + ev)$ [a] $+ (n_1 - 1) * (\lvert Cpr \rvert * pr + \lVert Cpr \rVert * \lvert Inf_i \rvert * (pr + ev))$
T_2	$\lvert T_1 \rvert * (pr + ev)$
T_3	$\lvert T_2 \rvert * pr + \lVert T_2 \rVert * pr$
T_4	$\lVert T_3 \rVert * (lev + lea / \lVert Cpr \rVert)$
T_5	$\lvert T_4 \rvert * (pr + ev)$
T_6	$\lvert T_5 \rvert * pr + \lVert T_5 \rVert * pr$
T_7	$\lvert Cpr \rvert * pr + \lVert Cpr \rVert * pr$
T_8	$\lVert T_7 \rVert * (lev + lea / \lVert Cpr \rVert)$
T_9	$\lvert T_8 \rvert * (pr + ev)$
T_{10}	$\lvert Inf' \rvert * pr + \lVert Inf' \rVert * pr$
T_{11}	$\lVert T_{10} \rVert * (lev + lea / \lVert Cpr \rVert)$
T_{12}	$\lvert T_{11} \rvert * (pr + ev)$
T_{13}	$\lvert Cpr \rvert * pr + \lVert Cpr \rVert * \lvert T_{11} \rvert * (pr + ev)$ [b]
T_{14}	$cost(Exp(T_9)) + (n_2 - 1) * cost(Exp(Inf_i'))$
T_{15}	$\lvert T_{14} \rvert * (pr + ev)$

[a] *Inf*, *Ins*, *Cpr*, and *Cpn* stand for *Influencer*, *Instrument*, *Composer*, and *Composition*, respectively

[b] these four operations constitute the fixpoint equation $Exp(Inf')$

Figure 7: The Cost of PTs of Figure 4

The next step optimizes predicate nodes and generates a "locally" optimal PT for each one of them; the *generative* approach builds several tentative PTs and chooses the least costly among them; the **Sel** and **EJ** nodes are generated at this step and the resulting PT resembles that of Figure 4.(i). The cost of this solution is sketched in the top part of Figure 7.

To simplify cost computation, we assume that no access structure other than the path indices are available, instances of sub-objects are not clustered close to the owner object, and no materialization of a PT node result occurs. Thus, we have,

$access\_cost(C_i,P) = \lvert C_i \rvert * pr$	$eval\_cost(C_i,P) = ev$ [a]
$access\_cost(C_i) = \lvert C_i \rvert * pr$	$nbtuples(C_i,P) = \lVert C_i \rVert$
$access\_cost(C_i,C_j) = pr$	$nbpages(C_i,P) = \lvert C_i \rvert$
$nbleaves(index) = lea$	$nblevels(index) = lev$

[a] pr, lea, lev, and pr are constants

Then, the PT of Figure 4.(i) is transformed into that of Figure 4.(ii) by **transformPT** which starts by applying the **filter** action. The cost of the obtained solution is sketched in the second part of Figure 7. Reoptimization, performed by **randOptimize**, tries to improve the PT of Figure 4.(ii) by considering unused indices (none in this case) or changing join permutations captured by the EJ nodes. As the final step of **transformPT**, the solutions of Figures 4.(i) and 4.(ii) are compared. The sketched costs clearly show that

the PT of Figure 4.(ii) is more costly than that of Figure 4.(i). Pushing selection through recursion in this example is not worthwhile.

5 Conclusion

In this paper, we have proposed a comprehensive approach for optimizing object-oriented recursive queries. Because the traditional heuristics proposed in deductive DBs cannot be unconditionnaly applied to conceptual queries when objects and recursion are involved, we argued for their use in the presence of a cost model. In particular, the dichotomy between rewriting and optimization needs to be revised. In our approach, execution plans are modeled by Processing Trees (PTs) which refer to physical entities and associated cost estimates, and the optimizer actions are expressed over PTs. Thus, we are able to estimate the cost of the effect of each transformation.

The main contributions of the paper are the following. Contrary to rewriting approaches [KM90], our query graphs enable to simultaneously optimize overlapping paths without any additional rewriting. This is due to the ability of using several variables along the same path of the query graph. As the optimizer actions are expressed over physical entities, and not conceptual ones, we are able to directly measure their impact based on a cost model. By supporting varying optimization granules, we are able to independently optimize subproblems (e.g., a path or a select-project-join) and then use the results in the optimization of a larger granule (e.g., a recursive expression). Our approach enables increasing the optimization opportunities by a better integration of rewriting and optimization. Thus, it guarantees optimality when investigating solutions where selective operations, i.e., selection and join, are pushed through recursion. In particular, pushing join through recursion has not been previously explored by optimizers.

We believe that many open problems in optimization can be tackled using this approach. For example, distributing union over join and vice-versa, that is not typically examined because of the undesirable increase in the search space [Se79, KZ88]. We are able to efficiently explore this transformation due to the ability of changing the optimization granularity and corresponding search strategy.

An optimizer prototype using this approach is currently being implemented at INRIA in C++ as part of the DBS3 project. Most of the rules herein proposed have been implemented. As usual, those for recursive queries are applied at a previous rewriting step but we plan to change the implementation of the optimizer to make it conform to the approach herein proposed. The cost model of the implemented optimizer is more general than the sketch shown in this paper, because it takes parallelism into consideration. Both enumerative and randomized search strategies are supported.

6 Acknowledgements

The authors are very grateful to Eric Simon and Guy Lohman for their detailed comments.

References

[AK89] S. Abiteboul, P. Kanelakis: "Object Identity as a Query Language Primitive", *SIGMOD* 1989.

[BR86] F. Bancilhon, R. Ramakrishnan: "An Amateur's Introduction to Recursive Query Processing Strategies", *SIGMOD* 1986.

[BCD89] F. Bancilhon, S. Cluet, C. Delobel: "Query Languages for object-oriented database systems: the O2 proposal", *DBPL* 1989.

[CS90] R.G.G. Cattell, J. Skeen: "Engineering Database Benchmark", *ACM TODS* 1990.

[GV92] G. Gardarin, P. Valduriez: "ESQL2: an Extended SQL2 with F-logic semantics", *Proc. IEEE Data Engineering* 1992.

[IC90] Y.E.Ioannidis, Y. Cha Kang: "Randomized Algorithms for Optimizing large join queries", *SIGMOD* 1990.

[KM90] A. Kemper, G. Moerkotte: "Advanced Query Processing in Object Bases Using Access Support Relations", *VLDB* 1990.

[KL86] M. Kifer, E.L. Loziinski: "A Framework for an Efficient Implementation of Deductive Database Systems", *Advanced Database Symposium* 1986.

[KZ88] R. Krishnamurty, C. Zaniolo: "Optimization in a Logic Based Language for Knowledge and Data Intensive Applications", *EDBT* 1988.

[LV91] R.S.G. Lanzelotte, P. Valduriez: "Extending the Search Strategy in a Query Optimizer", *VLDB* 1991.

[LVZC91] R.S.G. Lanzelotte, P. Valduriez, M. Ziane, J.P. Cheiney: "Optimization of Nonrecursive Queries in OODBs", *DOOD* 1991.

[MS86] D. Maier, J. Stein: "Indexing in an Object-Oriented DBMS", *Workshop on Object-Oriented Database Systems* 1986.

[Se79] P.G. Selinger et al.: "Access Path Selection in a Relational Database Management System", *SIGMOD* 1979.

[VKC86] P. Valduriez, S. Khoshafian, G. Copeland: "Implementation Techniques of Complex Objects", *VLDB* 1986.

[Va87] P. Valduriez: "Join Indices", *ACM TODS*, Vol. 12, N. 2, 1987.