# Loop Optimizations for Acyclic Object-Oriented Queries

Vasilis Samoladas Daniel P. Miranker

The University of Texas at Austin
Department of Computer Sciences
Taylor Hall 2.124
Austin, TX 78712-1188
{vsam,miranker}@cs.utexas.edu
Tel: (512)–471–9541

## Abstract

Nested loop execution of object-oriented queries retains the promise of maintaining the full generality of the object paradigm, independent of the specifics of any single object model. Thus, from this starting point we have developed an object-oriented query optimizer and execution engine. The methods, developed to date for only acyclic queries, augment nested loops structures with a simple marking mechanism such that unnecessary loop iterations are not repeated. In the case of acyclic queries, the executions are asymptotically optimal. In contrast to optimal query methods based on semijoin reductions our method involves no preprocessing step and thus avoids the extra I/O associated with semijoins and prevents the formal benefits of semijoin reduction from appearing as a practical improvement. Empirical results comparing our query environment with a commercially available product demonstrate significant performance improvement.

## 1 Introduction

We develop a technique for the optimization and computation of acyclic multi-join queries that deviates from mainstream techniques. The basis of the method is the mapping of a query and a database schema to a query graph, omitting an algebraic representation. The system develops a query plan in conjunction with a search for a minimum-cost spanning tree of the graph. We are developing this approach in the context of Object-Oriented Databases (OODBs) where the lack of universally accepted algebraic and semantic models makes this approach particularly advantagous.

It is convenient to think of our approach as a method of optimizing nested loops. Despite the commonly held belief that there is little opportunity to optimize nested loops [20] we present simple bookkeeping methods with which to augment loops and that eliminate many unnecessary loop iterations. For acyclic queries, we have achieved better performance than a traditional query evaluation engine used in a commercial OODB. In fact our approach captures all the formal advantages of query optimization using semijoin reductions and cures the practical disadvantages. In particular, we are able to compute a representation of an acyclic $n$-way join in optimal time and space, $O(nk^2)$, where $k$ is a bound on the size of collections. A key element of the technique is that it performs reductions "on-the-fly". The output can be materialized either during this computation, or later, in time linear with the size of the output. Thus, the additional pass through the data normally associated with semi-join reduction can be avoided, skipping the additional I/O that limits the use of semi-join reductions.

Query plans derived from nested loops have other advantages. We will show that such query plans are easy to produce and execute, producing only a small amount of code per query. Good locality and caching behavior should be witnessed, as objects are accessed in a depth-first manner (in terms of the object-schema) as opposed to the breadth-first manner of relational primitives. This becomes important in client-server OODB systems, where query processing usualy takes place on the client, rather than on the server. The asymptotic impact on cost of on-the-fly reduction results in very fast execution for acyclic queries.

An important consideration in our work has been the development of techniques that would be widely applicable to OODBs. We adopt instantiation semantics for abstract conjunctive queries. A *successful query instantiation* is an assignment of object instances to the variables of the query, such that all navigational and propositional elements are satisfied. Formally, we understand the *result* of such a query to be the set of all successful query instantiations. These definitions allow us to exploit a cursor-based interface to access collections - again allowing us to ignore the vagaries of individual systems. We will emphasize that our technique is orthogonal to, and can incorporate most of the existing elements and techniques of query processing; access path selection, attribute and path indices, other join algorithms etc.

## 2 Definitions

In part, the added difficulty in the optimization and evaluation of OOQs is the integration of both the purely propositional elements that define a query and the physical elements that can be exploited to improve query performance. These, so called, physical elements manifest in path expressions which can be used to navigate through the compositions used to form complex objects. In this section we develop the definition of a query graph for object-oriented queries such that both the propositional and navigational elements are represented in a unified fashion.

### 2.1 Some OOQ Definitions

For the sake of brevity, we provide intuitive definitions for a number of query-related terms. Consider the schema of Fig.1.

A simple query in an SQL-like language might be the following:

```
QUERY Example1 =
   SELECT <...>
   FROM s IN Student::extent,
        c IN s.courses
        p IN Professor::extent
   WHERE
        s.FirstName == "John"
     && Q(c,p) ;
```

The `FROM` clause declares a number of range variables. Two of these, `s` and `p`, are defined on base collections. These we will call *independent variables*. Variable `c` however, is defined on a "nested" domain. Such a variable will be called a *dependent variable*, and we shall use the term *domain variable* for the variable that defines its domain. In the above example, `s` is the domain variable of `c`.

Navigational elements of the query, i.e. the domains and dependencies of variables with respect to the schema, are defined in the `FROM` clause.

The `WHERE` clause of the above query is a boolean conjunction of two predicates. The first of these predicates refers to a single variable. Predicates that refer to a single variable will be called *s-predicates* (for "selection"). The second of these predicates refers to two variables. Such predicates will be called *j-predicates* (for "join"). In this paper, we restrict to predicates of arity at most 2.

The `WHERE` clause defines the *propositional elements* of a query. We restrict our attention to queries where the `WHERE` clause is a conjunction of s-predicates and j-predicates. Such queries will be called *abstract conjunctive queries.*

### 2.2 An OOQ Graph

For an abstract conjunctive query $Q$, we define the *query dependency graph* (QDG) $G(Q) = (V, N, P)$ as follows: $V$ is the set of nodes, one for each variable of the query. Let $u, v$ be nodes. $(u, v) \in N$ is a *directed* arc from $u$ to $v$, iff $u$ is the domain variable of $v$, where of course $v$ is a dependent variable (we identify nodes with the corresponding variables from now on). $(u, v) \in P$ is an *undirected* arc between $u$ and $v$ iff there exists at least one j-predicate naming both $u$ and $v$. The query dependency graph for query `Example1` is shown in Fig.2.

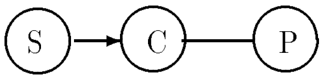

Figure 2: The query dependency graph $G$(Example1).

The directed arcs in a QDG represent the navigational elements of the query, whereas the undirected arcs denote propositional elements. Notice that between two nodes $u$ and $v$ there may exist both a directed and and undirected arc.

Let $G = (V, N, P)$ be a QDG. We define $G' = (V, N \cup P)$ to be the *lumped graph.* $G'$ is a regular, undirected graph.

**Definition 1** *$G$ is* **cycle-free** *iff the lumped graph $G'$ is acyclic (i.e. $G'$ is a tree).*

Let $u \in V$ be a node of $G = (V, N, P)$.

**Definition 2** *$u$ is a* **potential root** *iff there exists a simple pseudo-directed path (i.e. one respecting the direction of arcs in $N$, even in the presense of both a directed and a directed arc between two nodes) from $u$ to every other node of $G$.*

**Definition 3** *$G$ is* **dependency-directed** *iff there exists a node $u$ which is a potential root.*

We define acyclicity of a query $Q$ in terms of the QDG $G$. In particular, $Q$ is acyclic iff $G$ is *cycle-free* and *dependency-directed.* Of these two requirements, the first is natural for acyclicity. The second requirement, although not directly related to acyclicity, is however useful. To see why, we must give a final definition. Let the nodes of $G$ be ordered in a *depth-first* fashion, $v_1, \ldots, v_n$,

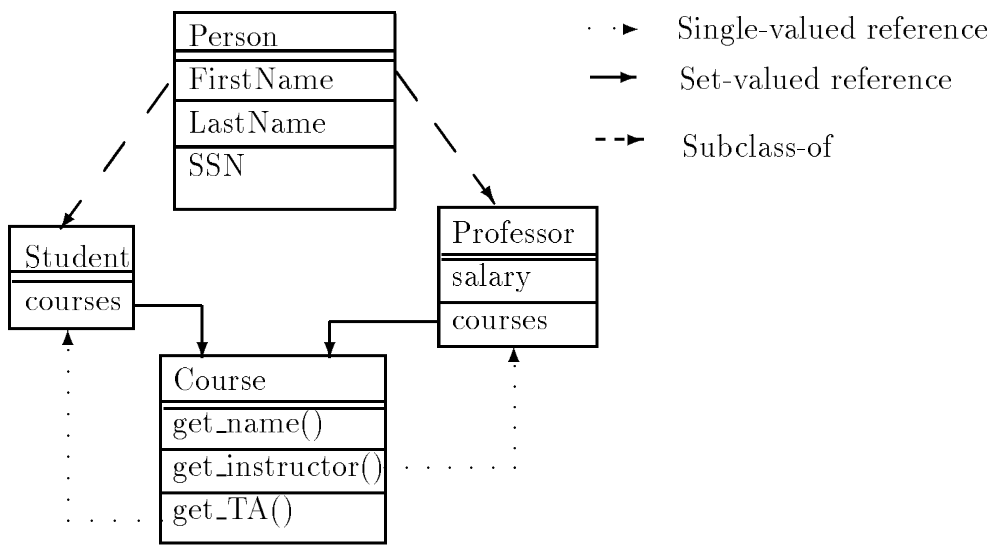

Figure 1: A simple OODB schema, consisting of 4 classes, 2 of which have nested collection attributes.

*where $v_1$ is a potential root.* We then refer to $G$ as an **ordered query dependency tree** (OQDT). Dependency-directedness is a necessary and sufficient condition for any depth-first ordering to have the following property:

**Proposition 1** *Let $X$ be a variable of an OQDT $G$. If $X$ is a dependent variable, then its domain variable is its parent in $G$.*

This proposition suggests that a backtracking depth-first traversal of the OQDT will respect the navigational elements of the query, i.e. will instantiate a domain variable before it attempts to instantiate the dependent variable.

## 3 Optimization of Nested loops

We prefer a new term, *naive nested-loops* for the construction commonly known as nested-loops evaluation. Naive nested-loops is the simplest form of depth-first evaluation. More intelligent forms of depth-first query evaluation have been around for a long time. Perhaps, the most familiar instance is the Prolog execution engine (see Ullman[23] for an extensive treatment of top-down evaluation).

The easiest optimization for depth-first query evaluation is intelligent backtracking. Naive nested loops can be thought of as naive backtracking. The variables of a query are ordered so that for each dependent variable, its domain variable precedes it in the ordering. Consider the OQDT of Fig.3. Suppose that we attempt to instantiate $X_4$ and fail to instantiate an object (because no object would satisfy the j-predicates with $X_1$). Naive backtracking –see Fig.3(b)– returns to $X_3$ and tries another instantiation of it, before retrying to instantiate $X_4$. However, instantiation of $X_4$ will fail again. The cause of failure is not the current instantiation of $X_3$, but that of $X_1$. In other words, the current instantiation of $X_1$ does not join with any value in $X_4$'s domain, thus it will not appear in the result. Naive backtracking eventually does the right thing, but with a high performance penalty. By employing *intelligent backtracking*, we backtrack to $X_1$ –see Fig.3(c), and we save an order of magnitude of extraneous work.

Another optimization of backtracking query execution involves recording information on the success or failure of a computation, so that the computation won't have to be repeated in subsequent iterations. The recorded information amounts to a marking of objects. We call this optimization *marking*, and comes in two variants, reduction marking and support marking.

Consider the join between $X_2$ and $X_3$ in the query of Fig.3. Assume that we have instantiated $X_2$ with object $x_2$ and we proceed to instantiate $X_3$. It is possible that no object of $X_3$'s domain joins with $x_2$. In this case, we mark $x_2$ as a failing instantiation, so that if $x_2$ becomes instantiated in the future, the algorithm will not have to repeat the failing computation. We call this optimization *reduction marking.*

We can also record successful computations. Consider again the previous scenario, only this time $X_3$ does successfully join with $X_2$. We can mark $x_2$ as a successfull instantiation. Then, if $x_2$ is instantiated again in the future, we can defer instantiating $X_3$, and try to instanti-

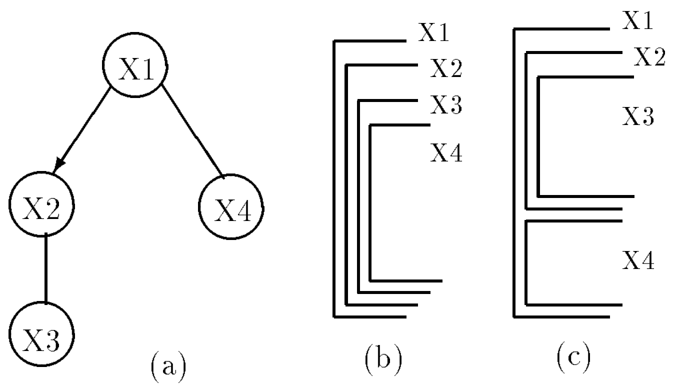

Figure 3: Intelligent backtracking: (a) query dependency graph (b) naive backtracking (c) intelligent backtracking

ate $X_4$ first. Since we know that instantiation of $X_3$ will succeed, there is no point in performing it eagerly, and maybe having $X_4$ fail. If $X_4$ does also succeed, we can come back to $X_3$ and complete the processing. We call this optimization *support marking.*

The ideas of marking and intelligent backtracking can be combined, and lead to very powerful optimizations of nested-loops query evaluation. In fact, these two kinds of optimizations, if applied correctly, are sufficient to guarantee asymptotically optimal performance in the computation of the *first solution* on acyclic queries [5]. In the following sections we present a specific algorithm that uses these ideas to deliver asymptotically optimal performance on acyclic queries, and discuss the database-specific issues that are involved, most notably implementation in an object data model and I/O performance.

## 4 Evaluating Acyclic Queries

In general, determining the set of successful query instantiations of an acyclic query takes exponential time in the worst case, simply because the size of the result $S$, i.e. the number of successful instantiations, can be exponential to the size of the database and the size of the query. Because of this, the lower bound is $\Omega(nk^2 + S)$.

However, it is possible to *represent* the result of an acyclic conjunctive query in polynomial space. Let $Q$ be an acyclic conjunctive query, and $G(Q)$ be an OQDT. Let $X_i$ and $X_j$ be variables of $Q$ such that $X_i$ is the parent of $X_j$ in the OQDT. Let $x_i$ be an object that instantiates $X_i$ in some successful query instantiation. With $x_i$ we associate a set $x_i.L_j$ of objects. An object $x_j$ belongs to $x_i.L_j$ iff for some successful query instantiation, $x_i$ instantiates $X_i$ and $x_j$ instantiates $X_j$. Also, define $L_1$ to be the set of all objects $x_1$ that instantiate $X_1$ (the root of the OQDT) in some successful query instantiation. Let these sets be called *goodlists*. Then, the result of the query can be enumerated by straight-forward application of naive nested loops.

```
**foreach**( $X_1 \in L_1$)
**foreach**( $X_2 \in X_1.L_2$ )
...
**foreach**( $X_i \in X_{parent(i)}.L_i$ )
...
**foreach**( $X_n \in X_{parent(n)}.L_n$ )
{
  // process instantiation
  Emit_Successful_Instantiation($X_1, \ldots, X_n$);
}
```

The number of iterations will be equal to the size $S$ of the query result. So, the computation of the goodlists of a given query can be viewed as optimized query evaluation, prior to naive nested-loop enumeration of the result. This is not a contradiction with our claim in the introduction that our technique does not require preprocessing, for the following two reasons. First, in semi-join reduction techniques, semi-join reduction is followed by a join phase, which in fact will compute the output. Our scheme does *not* require that. The nested loops above just *enumerate the result* of the query, which –for all practical purposes– *is* the set of goodlists. Second, this enumeration is not necessarily an after-step, but can be dispersed in the computation of the goodlists, in order to eliminate I/O overhead.

Although the above definition of goodlists is set-theoretic, it has a direct implementation in OODBs, by exploiting *OID semantics*. So, in practice, goodlists are sets of OIDs. Further, as the name suggests, they can be implemented as lists. More details about the implementation will follow.

The space requirements for goodlists are polynomial in the size of the database. If no domain contains more than $k$ objects, the worst-case space complexity is $O(nk^2)$

(there are $O(nk)$ objects in the database, and each object can appear in at most $k$ goodlists). Additionally, the total space is bounded by $O(nS)$, which can be much better than $O(nk^2)$ for small values of $S$. Notice that these space complexities are based on the fact that goodlists can be implemented as lists of OIDs to objects. Thus, each goodlist will require space proportional to the number of goodlist elements, independent of the size of the objects pointed at.

Under this context, the computation of query evaluation is reduced to the efficient computation of the goodlists for a given query and OQDT. We will describe an optimal backtracking algorithm for the computation of goodlists, postponing any system-related issues for the following sections. The algorithm incorporates marking and intelligent backtracking, as they were described in the previous section. Marking is implemented by associating a *status marking* with each object. [1] We assume a generic cursor interface, where we have a cursor per query variable. We will use the term "instantiates" to refer to cursor operation, assuming that all navigational and/or propositional constraints with the parent cursor's instantiation are enforced.

At any time, the status marking of an object is one of **unknown**, **supported**, **completed** and **deleted**. The significance of these markings is the following:

**deleted** An object $x$ instantiating $X$ is marked **deleted** as the result of reduction marking. This will happen if the algorithm fails to instantiate some child of $X$. Once an object is marked as deleted, it is (conceptually) removed from the domain of $X$.

**supported** An object $x$ instantiating variable $X$ becomes **supported** as a result of support marking. Intuitively, supportedness means that the backtracking computation is known to succeed in the subtree of $X$ with respect to $x$, and can be deferred in future instantiations of $x$, in the spirit of the previous section.

**completed** An object $x$ instantiating variable $X$ becomes **completed** when all the goodlists associated with $x$ have been fully computed. Notice that only **supported** objects can become **completed**. If $X$ is a leaf, then $x$ is initialized to this marking (because it is supported and it does not have any goodlists to compute).

**unknown** If nothing is known about an object, it is marked **unknown**. Initially, all objects —except those instantiating leaf nodes— are marked **unknown**.

[1] Technically, a status marking is associated with a pair (OID,variable), as some object may instantiate multiple varibales. This technicality is important in the implementation, but not in the presentation of the technique

It is most convenient to understand the functioning of the algorithm with respect to the root variable. For each instantiation $x_1$ of the root variable $X_1$, the algorithm executes a top-down phase and a bottom-up phase.

In the top-down phase, the algorithm attempts to determine whether $x_1$ is supported, by finding a first successful query instantiation. This computation is recursively reduced to instantiating the children of $X_1$, and then computing supportedness of these instantiations. This evaluation is done depth-first, by taking advantage of previous support marking, and applying intelligent backtracking and reduction marking when appropriate.

If a successful query instantiation is found, object $x_1$ is marked **supported** and a bottom-up phase begins, that will result in computing the goodlists of $x_1$ and marking it **completed**. In the case of failure, $x_1$ is marked **deleted** and the algorithm attempts to get the next instantiation of $X_1$.

The bottom-up phase is straightforward. Its purpose is to compute goodlists of current instatiations. For the query of fig.3(a) , suppose that $x_1$ has been marked supported, right after a successful instantiation of $X_4$. The bottom-up phase begins by building $x_1.L_4$ (one of the goodlists of $x_1$), and then continuing backwards, i.e. building the goodlist for the current instantiation of $X_2$, and then getting another instatiation for $X_2$ that joins with $x_1$, and so on.

This algorithm can be implemented simply, by splitting up the processing into three operators, implemented as co-routines, called ADVANCE, BACKTRACK and RETREAT. These co-routines encompass the logic of the iteration. ADVANCE implements the top-down phase and support marking. BACKTRACK implements reduction marking. RETREAT implements the bottom-up phase. Our implementation reflects the ususal architecture of the back-end of a query optimizer. The three operators,ADVANCE, BACKTRACK and RETREAT, can be treated as templates, instantiated together with specific join algorithms during code generation. The query program reflects a pipeline structure.

## 4.1 Asymptotic complexity analysis

Let $X$ and $Y$ be variables in a query, such that $X$ is the parent of $Y$. For objects $x$ and $y$ in the domains of $X$ and $Y$ respectively, let $x \to y$ denote the event that $y$ instantiates $Y$ and $x$ instantiates $X$.

With respect to the complexity of the algorithm, the following proposition applies:

**Proposition 2** *During the execution of the algorithm, for variables $X$ and $Y$, such that $X$ is the parent of $Y$,* *and for some instantiations $x$ of $X$ and $y$ of $Y$ respectively, the number of times that $x \rightarrow y$ will occur is at most twice. Further, for any given $x$, there is at most one $y$ for which $x \rightarrow y$ will occur more than once.*

**Sketch of proof:** The first time that $y$ instantiates $Y$ while $x$ instantiates $X$, will either be during a top-down phase, or during a bottom-up phase w.r.t. $X$.
If during a bottom-up phase, then $y$ will be inserted into the appropriate goodlist, and $x$ will eventually be marked **completed**. Thus, if $x$ instantiates $X$ in the future, none of $X$'s descendants (including $Y$) will be iterated over.
If during a top-down phase, three outcomes are possible: (a) $x$ will be **deleted**, (b) $y$ will be **deleted**, (c) $x$ will become **supported**. Cases (a) and (b) are trivial. If the case is (c), then it is possible in the future that $x$ and $y$ will instantiate $X$ and $Y$. But, because $x$ is already **supported**, it will have to be during a bottom-up phase. So, in this case, we may have $x$ and $y$ being instantiated twice.□

The complete proof is by quite straightforward but elaborate case analysis, and is omitted. By virtue of this proposition, we compute the asymptotic complexity of the algorithm in a straightforward way: There are exactly $n-1$ parent-child pairs of variables. Each of these pairs of variables defines $O(k^2)$ pairs of instantiations. By the above proposition, each of the j-predicates between these two nodes will be applied $O(k^2)$ times. So, there will be at most $O(nk^2)$ j-predicate invocations. Also, from the proposition we infer that each object will be instantiated $O(k)$ times. Then, there will be at most $O(nk^2)$ s-predicate applications. Finally, each operator performs constant work for every instantiation (adding an element to the head of a goodlist is constant work). Thus, the overall asymptotic complexity of the algorithm is $O(nk^2)$. Here, we do not include the inherent complexities of the predicate evaluations, we are just bounding the overall number of predicate applications.

## 5 Implementation issues

We have chosen to present the concept of intelligent backtracking with learning in a system-independent manner. However, if this technique is applicable to database systems, it must –and will– be supported by arguments of its suitability in and environment where I/O costs are the major performance burden. Also, it must be shown to be integrable with real OODBMS systems. We will first then address a number of implementation issues.

### 5.1 Markings and goodlists

For a given query, some object may instantiate more than one variable. So, an implementation would associate a marking and a number of goodlists with a pair of the form (OID,variable), instead of just an OID. However, markings and goodlists, because of their use, are transient values, and need not be stored on secondary storage. In any case, a marking can be stored in just 2 bits, and in general does not pose a problem. Goodlists on the other hand have the semantics of sets. An important point is that the only operation performed on the goodlists during the execution of the algorithm is insertion. Thus, it is possible to allocate space on the secondary storage and store them there in an efficient manner, by some simple batch-writing technique.

In our implementation, markings and goodlists were stored in transient memory. The association of OIDs to this data, was implemented by simply adding a "pointer to transient memory" attribute to every object in our schema. During the course of a transaction, these pointers pointed to small arrays of markings and goodlists. This is not the best strategy, because it involves accessing the object to obtain its marking. More sophisticated techniques should be used in heavyweight database applications.

### 5.2 Selections and indices

In our scheme, domains of independent variables with $s$-predicates were materialized as new collections (except for the domain of the root). This simplistic strategy had some advantage in sequential scans, by physically deleting objects from these collections, instead of just marking them **deleted**. Selection predicates were applied during this stage, using any available indices to speed up the process. This is analogous to the well-known heuristic of pushing selections as low as possible in an algebraic query expression.

### 5.3 Joins and query programs

Although we have based our algorithm on a generic cursor interface, we anticipate the practicality of more sophisticated join algorithms. A careful study of our technique reveals that, with modest alterations, most join algorithms can be integrated with our algorithm. The performance savings of our approach do not come from the use of specialized access techniques, but from intelligent scheduling during what can considered as a pipelined query execution. Once suitable join algorithms are selected by the optimizer, the code generator will produce specialized versions of the Advance, Retreat and Backtrack operators, one version per variable.

### 5.4 Data access

Apart from access mechanisms used in relational systems, a number of OODB-specific techniques have been proposed, most notably path indices and function (method) materialization. As with all other access mechanisms we know about, our technique can take full advantage of such support from the storage manager.

#### 5.4.1 Query optimization

A typical query optimizer architecture is shown in fig.4. A declarative query is parsed and type-checked, and then

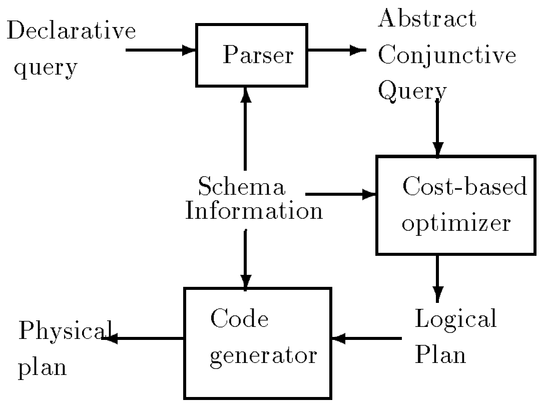

Figure 4: Architecture of a query optimization engine

translated into an abstract conjunctive query. This is fed into an optimizer. For reasons to be discussed, a cost-based optimizer is most suitable for our acyclic query execution approach. The output of this stage is a logical plan for execution, consisting of an ordering of the variables, a set of access paths, and a selection of join algorithms. The logical plan is input to the code generation stage, which is responsible for producing suitable operators on a per-variable basis, and perform code-related optimizations. The code generation would probably target an abstract instruction set at the lowest possible level of the data management system.

The development of a query optimizer, whether cost-based or rule-based, requires an analytical cost model, to derive conclusions on the performance of alternate execution plans. We provide a minimal analytical model, which was shown to be pretty accurate in our experiments.

Consider a query of $n$ variables. It will involve $n-1$ joins between these variables. Here, we are using the term “joins” liberally, to denote both data access on navigational elements (i.e. accessing the data of “precomputed” joins), and on propositional elements (i.e. selections and actual joins). By $J_i$ we denote the joins in a logical query plan. Then, the overall cost of the query is

$$Cost = C_L + \sum_{i=1}^{n-1} Cost(J_i)$$

where $C_L$ is the cost of storing the goodlists. This cost of storing goodlists depends on the ordering, but –for reasons mentioned before– it is negligible with respect to the cost of the joins.

This expression for the cost is a consequence of prop.2. What is important is that the cost expression does not have a recursive form. The most important choice of an optimizer is the choice of the root (as it may constrain the options for some of the join algorithms). Once a root is selected, joins and access paths of minimum cost can be selected on a per-variable basis. Since there are at most $n$ potential roots, optimization in polynomial time is feasible.

## 6 Experimental results

We have implemented a simple query engine along the lines described so far. We used the ObjectStore commercial OODBMS, running on an HP 900/735 workstation, with remote disk access over a local area network. Queries are input in the conjunctive abstract query form. We only implemented block-nested-loop joins and indexed-nested-loop joins. Cost-based optimization is manual, so that we perform evaluation experiments of the cost model. Due to the unavailability of a benchmark suite for OODB multijoin queries, we synthesized random data for our experiments. We present two groups of results. The first group involves 3 queries on databases of different sizes, that fit into the main memory. For these queries, indexes were unusable, because the j-predicates involved arbitrary function calls. These queries test the efficiency of the approach in lieu of any acceleration mechanisms. The second group of results involves 3 queries over a large database, with full index support, and equalities as join predicates. These results demonstrate the performance of the algorithm in traditional, data-intensive queries.

Table 1 presents the results of the first group of experiments, shows the performance characteristic of 3 queries on 3 synthesized databases of different sizes, with the schema of fig.1. Table 2 presents the second group of results. The database was twice the size of physical memory. The schema was constructed after the Wisconsin benchmark paradigm, with integer unique or uniformly-distributed values in the various attributes.

For comparison, we used ObjectStore's internal query facility to perform similar query processing on our data. As reported in [17], the ObjectStore query facility executes neted loops with intelligent backtracking, and optimizes access paths, using any available indexes. Because

| **Small database: 10000 students, 100 professors, 500 courses** | | | |
|---|---|---|---|
| | Query 1 | Query 2 | Query 3 |
| CPU time (sec) | 142 | 2.6 | 153 |
| Total time (sec) | 156 | 16 | 166 |
| CPU Utilization | 91.4% | 26.1% | 92.3% |
| Total of goodlist nodes | 132 | 10 | 3419 |
| Size of output $S$ | 44 | 5 | 1498 |
| ObjectStore CPU time (sec) | 216 | 3.0 | 603 |
| Cost model estimate | 151.8 | 1.2 | 161.7 |
| $\tau_i$ estimate ($\mu$sec) | 28.1 | 107.5 | 28.4 |
| **Medium database: 10000 students, 200 professors, 1000 courses** | | | |
| | Query 1 | Query 2 | Query 3 |
| CPU time (sec) | 262 | 2.7 | 298 |
| Total time (sec) | 283 | 22 | 310 |
| CPU Utilization | 92.5% | 34.7% | 96.1% |
| Total of goodlist nodes | 303 | 4 | 6733 |
| Size of output $S$ | 101 | 2 | 2997 |
| ObjectStore CPU time (sec) | 412 | 3.2 | 1156 |
| Cost model estimate | 306.3 | 1.2 | 313.2 |
| $\tau_i$ estimate ($\mu$sec) | 25.5 | 85.3 | 28.5 |
| **Large database: 30000 students, 1000 professors, 5000 courses** | | | |
| | Query 1 | Query 2 | Query 3 |
| CPU time (sec) | 3975 | 12.5 | 4965 |
| Total time (sec) | 4033 | 44 | 5000 |
| CPU Utilization | 98.5% | 28.4% | 99.2% |
| Total of goodlist nodes | 1587 | 6 | 32787 |
| Size of output $S$ | 531 | 3 | 14994 |
| ObjectStore CPU time (sec) | 9876 | 14.5 | 18796 |
| Cost model estimate | 4650 | 3.6 | 4953 |
| $\tau_i$ estimate ($\mu$sec) | 25.6 | 104.2 | 30.1 |

Table 1: Experiment results on three queries. Query 1 is a 3-way join, query 2 is simple unnesting, query 3 is a 4-way join with unnesting. The cost model estimate was based on the assumption that $\tau_i = 30\mu$sec

we incorporate marking, we expected to achieve better performance, and indeed our results indicate that. Notice that the ObjectStore query facility does not perform explicit joins. The result shown are for computation that is roughly analogous to the top-down pass of our algorithm (i.e. attempt to establish supportedness of instantiations of the root variable). The numbers for ObjectStore represent just that computation. On the contrary, the numbers for our algorithm represent the full computation. Additionally, we are fairly certain that the physical access paths generated by ObjectStore's query optimizer were better than those our optimizer generated. Even so, the results indicate that marking can introduce substantial savings to the execution time.

A few comments are in order. In the first group, the improvement in runtime by augmenting intelligent backtracking with marking is impressive. Also interesting is the minimal space overhead, which can be assessed as 8 bytes per goodlist node. Contrast that with the size of the output (in terms of number of successful instantiations) and with the large space overhead sometimes exhibited by set-oriented processing.

For the second group of results, observe how performance is affected by marking, even in the full presense of indices. Indeed, in query 1, which was large, we achieved better performance than ObjectStore, while actually computing more. In query 2, the great difference in performance came from reduction marking. The selectivities of this query were unusually low, and our approach took advantage of this easily, whereas intelligent backtracking by itself, was consumed in repeating the same failures over and over. An interesting point -no data shown here- had to do with the breakups of these times into I/O times and memory management times. Our implementation was much less sophisticated than ObjectStore's, in terms of resource management and index access. Thus, much of our CPU time was spent in memory allocation. ObjectStore spent minimal time in such tasks. This fact shows

| Large database: 48 Mbytes of data per query | | | |
|---|---|---|---|
| | Query 1 | Query 2 | Query 3 |
| Our CPU time (sec) | 115 | 19 | 3.9 |
| Total time (sec) | 199 | 35 | 9 |
| Utilization (%) | 58 | 54 | 38 |
| ObjectStore CPU time (sec) | 442 | 432 | 2.64 |

Table 2: Performance on a large, fully indexed database. Query 1 is a 4-way join, query 2 is a 3-way join, query 3 is a join between a base table and an unnested variable.

even greater savings of our approach. Finally, the depth first approach did indeed exhibit good locality, as shown by our utilization figures (the fact that we wasted CPU time inefficiently should be countered by the fact that we did not run on a local disk).

Most important for our optimizer was the verification of our cost model. As proposed, the cost model accurately computed the CPU cost of query execution. More precise modeling of I/O costs is definitely necessary in a real system, but it is dependent on the storage manager's implementation, and was outside the scope of this research. Below we present the specific naive cost model that we used; joins are computed either by nested loops of through indices. Assume a specific query of $n$ variables $X_i$, and an ordering of these variables. The following quantities are defined:

$s_i$ is the average size of the domain of $X_i$, i.e. the number of objects in Domain($X_i$) (for dependent variables, this is an average).

$D_i$ is the virtual domain size, i.e. the number of distinct objects instantiating $X_i$. For independent variables, $D_i = s_i$. For dependent variables, $D_i$ is at most the size of the unnesting, but can be much smaller.

$\sigma_i$ is the composite selectivity of s-predicates naming $X_i$.

$\theta_i$ is the composite selectivity of indices on $X_i$. If there are no indices defined on the domain $X_i$, $\theta_i = 1$.

$b_i$ is the average object size for objects in the domain of $X_i$.

$t_i$ is the number of scans performed on the domain of $X_i$. More precisely, it is the average number of times that each object instantiates $X_i$.

$\tau_i$ is the average access time for objects in the domain of $X_i$.

Of these quantities, $s_i, D_i, \sigma_i, \theta_i$ and $b_i$ would come from the database catalog. To simplify things, introduce $S_i$ to be the effective virtual domain size of $X_i$, as

$$S_i = \sigma_i \theta_i D_i$$

The average access time $\tau_i$ is strongly dependent on the storage management architecture and characteristics. For typical client-server OODBMS systems, where data is being fetched from the server and cached to the client, we use a formula of the following sort:

$$\tau_i = h\tau_m + (1-h)\tau_s b_i$$

where $h$ represents the cache hit ratio. This is a very coarse model of the I/O and predicate costs involved, but proved sufficient for the queries mentioned herein.

Based on the above quantities, the overall time cost $T$ of the query execution is given by

$$T = \sum_{i=1}^{n} t_i \theta_i s_i \tau_i$$

The value of $t_i$ is data dependent, since the algorithm is data-driven. However, from proposition 2 we can derive bounds for the value of $t_i$. Let $X_\alpha$ and $X_\beta$ be variables, $X_\alpha$ being the parent of $X_\beta$. Then,

$$S_\alpha \leq t_\beta \leq 2S_\alpha$$

Of course, $t_1 = 1$. Let $\mu$ be the repetition ratio, so that

$$t_\beta = \mu S_\alpha$$

Clearly, $1 \leq \mu \leq 2$. In practice, $\mu$ is very close to 1, and for some queries (depending on the query graph) it is exactly 1. Typical values of $\mu$ for large queries on uniformly random data sets range from 1 to 1.05.

## 7 Discussion

We claim the ideas presented in this paper are especially advantageous for the object data model. However, the same techniques have potential on relational database systems as well. One avenue may be to exploit semantic information derived from the declaration of foreign keys as a precomputation of support markings. The ultimate success in this context remains to be seen. Given the maturity and power of RDB technology these techniques will have to be evaluated in the context of better refined cost models and optimizations that are not applicable to OODBs. In addition, these loop optimization techniques are convenient with respect to the methods OODBs provide for iterating over collection. Similar interfaces to RDBs typically incur substantial overhead. Also, our approach handles in a uniform manner various aspects of OODB query processing that have been problematic, such as joins on method values, unnesting, and optimization.

It is often argued that explicit joins are not as important in OODBs as they are in RDBs, because in an intuitive schema they would be already "precomputed" in the form of nested collections. This is true for OODB applications that resemble traditional RDB applications. However, OODBs are becoming the platform of choice for a variety of new database applications, that are compute-intensive as well as data-intensive. These include CAD systems, distributed systems, and the Internet. In some of these applications, join queries between tens of collections are not uncommon. We particularly have in mind data mining queries which assemble complex barter opportunities among players on the internet.

Both RDB and OODB query languages support quantified (e.g. existential) predicates. In terms of execution, quantification is evaluated by iteration. Query strategies to optimize such iteration are important, and we should not ignore them in our approach. It so turns out that our algorithm can be extended in a straightforward manner to handle existential predicate evaluation, provided that the query graph of the predicate is acyclic. The extension would simply be to represent existential iteration by "hidden" variables in the query. For these variables, the bottom-up phase of the iteration should be omitted. In other words, goodlists will not be built for the instantiations of these variables, but markings will be recorded. Negation (i.e. universal quantification) can be supported, by simply reversing the meanings of **deleted** and **supported** markings for appropriate variables. We have not yet implemented this however, so we will not expand any further.

The obvious extension to this research is of course the extension of these techniques to non-acyclic queries, and to queries with query graphs that are not dependency-directed. For both of these directions there are obvious extensions, but we do not yet have concrete results. Consider however the following simple approach to queries with only a few (1 or 2) cycles: select a spanning tree of the query graph, and compute the goodlists for it. Then, iterate over the goodlists, and discard any instantiations that do not satisfy the predicates that are not part of the selected tree. Although this approach seems naive, it may be that for queries with only 1 or 2 cycles, and with intelligent backtracking in the iteration over the goodlists, it is a good strategy. This is because (a) the computation of the goodlists is very fast, (b) the space required is small (c) in many cases, careful selection of the spanning tree will reduce the successful instantiations (to be tested against the extra predicates) to a small number.

## 8 Related work

The techniques we present in this work, were derived from a recent result in Contraint Satisfaction (CSP). This well-studied AI problem roughly corresponds to the problem of non-emptiness of a relational database query. A search of the CSP literature reveals a number of results and techniques analogous to results and techniques in relational query execution. However, the transition from a CSP result to a database result is non-trivial at the least. In the light of this work, we emphasize that an analogous technique to semijoin reduction did exist in the CSP world, under the term Directed Arc Consistency. A recent development in this area has been the TreeTracker algorithm[5], from which we have derived the results of this work.

## 9 Conclusions and future work

We have developed and evaluated a new approach to OODB query execution, based on intelligent backtracking and a form of nested-loop optimization, which we call marking. We advocate the suitability of this approach to the object-oriented model, as it is algorithmic in nature, and poses very few semantic restrictions to the data model. We have applied these ideas to a solution to the important subproblem of acyclic query evaluation. The resulting algorithm is asymptotically optimal, and has experimentally been demonstrated to deliver excellent performance. The nature of the approach results in a simple but accurate cost mode; one that can be used to select a near-optimal execution plan in time quadratic to the size of the query. Based on our experimental data, we advocate the depth-first approach to query execution, and demonstrate it to alleviate some of the problems of performing unnesting on the data.

As a formal foundation, our approach relates to the Constraint Satisfaction Problem[11, 5]. Toward this goal we have developed a nearly uniform representation of elements of the object schema and the query in the form of a query graph. This form of a query graph retains the declarative form of a query. It follows from this graph definition that query evaluation can be defined with respect to a set of cursors, one cursor per vertex of the query graph, such that these cursors iterate over collections of objects. There is no restrictions on the implementation of the cursors. Thus, the approach simultaneously remains flexible with respect to physical access methods and may generalize over many differeent OODB data models and architectures.

The results reported are part of work in progress. We intend to investigate the generality of these techniques, by developing algorithms for more general problems than acyclic queries. Immediate goals are an algorithm for dependency-directed non-acyclic queries, a systematic handling of quantified predicates, and implementation under different OODBMS systems, to study system-related issues under different architectural choices.

## References

[1] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. The object-oriented database manifesto. In *Proceedings of the Int'l Conference on Deductive and Object-Oriented Databases*, Kyoto, Japan, December 1994.

[2] Francois Bancilhon, Claude Delobel, and Paris Kanellakis, editors. *Building an Object-Oriented Database System - The Story Of $O_2$*. Morgan Kaufmann Publishers, San Mateo, CA, 1992.

[3] Roberto J. Bayardo Jr. Enhancing query plans for many-way joins. Unpublished article.

[4] Roberto J. Bayardo Jr. and Daniel P. Miranker. Backtrack-bounded search in polynomial space. Unpublished article.

[5] Roberto J. Bayardo Jr. and Daniel P. Miranker. An optimal backtrack algorithm for tree-structured constraint satisfaction problems. *Artificial Intelligence*, 1994.

[6] Catriel Beeri, Ronald Fagin, David Maier, and Mihalis Yannakakis. On the desirability of acyclic database schemes. *Journal of the ACM*, 30(3):479–513, July 1983.

[7] Elisa Berino, Mauro Negri, Giuseppe Pelagatti, and Licia Sbattella. Object-oriented query languages: The notion and the issues. *IEEE Trans. on Knowledge and Data Engineering*, 4(3):223–237, June 1992.

[8] Elisa Bertino and Lorenzo Martino. *Object-Oriented Database Systems: Concepts and Architectures.* International Computer Science Series. Addison-Wesley, 1993.

[9] José A. Blakeley, W. J. McKenna, and G. Graefe. Experiences building the open oodb query optimizer. In *Proceedings of the SIGMOD COnference on the Management of Data*, pages 287–296, Washington, DC, May 1993. ACM.

[10] M. Carey, D. DeWitt, G. Graefe, D. Haight, J. Richardson, D. Schuh, E. Shekita, and S. Vandenberg. The exodus extensible dbms project: An overview. In S. Zdonik and D. Maier, editors, *Readings in Object-Oriented Databases*. Morgan-Kaufman, 1990.

[11] Rina Dechter. Constraint networks. in Encyclopedia of Artificial Intelligence, 2nd Ed., 1992.

[12] Goetz Graefe. Volcano-an extensible and parallel query evaluation system. *IEEE Trans. on Knowledge and Data Engineering*, 6(1):120–135, February 1994.

[13] Matthias Jarke and Jürgen Koch. Query optimization in database systems. *Computing Surveys*, 16(2):111–152, June 1984.

[14] Alfons Kemper, Christoph Kilger, and Guido Moerkotte. Function materialization in object bases: Design, realization, and evaluation. *IEEE Trans. in Knowledge and Data Engineering*, 6(4):587–608, August 1994.

[15] Michael Kifer, W. Kim, and Y. Sagiv. Querying object-oriented databases. In *Proc. of 1992 ACM SIGMOD*, pages 393–402, CA, USA, June 1992. ACM.

[16] José Meseguer and Xiaolei Qian. A logical semantics for object-oriented databases. *SIGMOD Record*, 22(2):89–98, June 1993.

[17] Jack Orenstein, Sam Haradhvala, Benson Margulies, and Don Sakahara. Query processing in the object-store database system. In *Proc. of 1992 ACM SIGMOD*, pages 403–412, CA, USA, June 1992. ACM.

[18] M. Tamer Özsu and José Blakeley. Query processing in object-oriented database systems. In W. Kim, editor, *Modern Database Management - Object-Oriented and Multidatabase Technologies*, pages 146–174. Addison-Wesley/ACM Press, 1994.

[19] M. Tamer Özsu, Adriana Munoz, and Duane Szafron. An extensible query optimizer for an objectbase management system. In *Proc. Fourth Int. Conf. on Information and Knowledge Management*, Baltimore, October 1995. (CIKM'95). (to appear).

[20] Hennie J. Steenhagen, Peter M. G. Apers, Henk M. Blanken, and Rolf A. de By. From nested-loop to join queries in oodb. In *Proceedings of the 20th VLDB Conference*, pages 618–629, Sandiago, Chile, 1994.

[21] Dave D. Straube and M. Tamer Özsu. Query optimization and execution plan generation in object-oriented data management systems. *IEEE Transactions on Knowledge and Data Engineering*, 7(2):210–227, April 1995.

[22] Stanley Y. W. Su, Mingsen Guo, and Herman Lam. Association algebra: A mathematical foundation for object-oriented databases. *IEEE Trans. in Knowledge and Data Engineering*, 5(5):775–798, October 1993.

[23] Jeffrey Ullman. *Principles of Database and Knowledge-Base Systems*. Computer Science Press, Inc., 1988.

[24] Scott Lee Vandenberg. *Algebras for Object-Oriented Query Languages*. PhD thesis, University of Wisconsin-Madison, 1993.

[25] Carlo Zaniolo. The representation and deductive retrieval of complex objects. In *Proc. of VLDB '85*, pages 458–469, Stockholm, 1985.