Paper Number AMERICA62

Query Optimization for XML*

Jason McHugh, Jennifer Widom
Stanford University
{mchughj,widom}@db.stanford.edu, http://www-db.stanford.edu

Abstract

XML is an emerging standard for data representation and exchange on the World-Wide Web. Due to the nature of information on the Web and the inherent flexibility of XML, we expect that much of the data encoded in XML will be *semistructured*: the data may be irregular or incomplete, and its structure may change rapidly or unpredictably. This paper describes the query processor of *Lore*, a DBMS for XML-based data supporting an expressive query language. We focus primarily on Lore's cost-based query optimizer. While all of the usual problems associated with cost-based query optimization apply to XML-based query languages, a number of additional problems arise, such as new kinds of indexing, more complicated notions of database statistics, and vastly different query execution strategies for different databases. We define appropriate logical and physical query plans, database statistics, and a cost model, and we describe plan enumeration including heuristics for reducing the large search space. Our optimizer is fully implemented in Lore and preliminary performance results are reported.

1 Introduction

The World-Wide Web community is rapidly embracing *XML* as a new standard for data representation and exchange on the Web [BPSM98]. At its most basic level, XML is a document markup language permitting tagged text (*elements*), element nesting, and element references. However, XML also can be viewed as a data modeling language, and a significant potential user population views XML in this way [Mar98]. Fortuitously, work from the database community in the area of *semistructured data* [Abi97, Bun97]—work that significantly predates XML—uses graph-based data models that correspond closely to XML. Thus, research results in the area of semistructured data are now broadly applicable to XML.

Semistructured data has been defined as data that may be irregular or incomplete, and whose structure may change rapidly or unpredictably. Although data encoded in XML may conform to a *Document Type Definition*, or *DTD* [BPSM98], DTD's are not required by XML. Due to the nature of information on the Web and the inherent flexibility of XML—with or without DTD's—we expect that much of the data encoded in XML will exhibit the classic characteristics of semistructured data as outlined above.

The *Lore* system at Stanford is a complete DBMS designed specifically for semistructured data [MAG+97]. Lore's original data model, the *Object Exchange Model* (*OEM*), is a graph-based data model with a close correspondence to XML. The query language of Lore, called *Lorel* (for *Lore Language*), is an expressive OQL-based language for declarative navigation and updates of OEM databases. We are in the process of migrating Lore to a fully XML-based data model and query language, but the results presented in this paper are changing very little as the migration occurs.

*This work was supported by Rome Laboratories under Air Force Contract F30602-96-1-0312 and by the National Science Foundation under grant IIS-9811947.

This paper describes Lore's query processor, with a focus on its cost-based query optimizer. While our general approach to query optimization is typical—we transform a query into a *logical query plan*, then explore the (exponential) space of possible *physical plans* looking for the one with least estimated cost—a number of factors associated with XML data complicate the problem. Path traversals (i.e., navigating subelement and reference links) play a central role in query processing and we have introduced several new types of indexes for efficient traversals through data graphs. The variety of indexes and traversal techniques increases our search space beyond that of a conventional optimizer, requiring us to develop aggressive pruning heuristics appropriate to our query plan enumeration strategy. Other challenges have been to define an appropriate set of statistics for graph-based data and to devise methods for computing and storing statistics without the benefit of a fixed schema. Statistics describing the "shape" of a data graph are crucial for determining which methods of graph traversal are optimal for a given query and database.

Once we have added appropriate indexes and statistics to our graph-based data model, optimizing the navigational *path expressions* that form the basis of our query language does resemble the optimization problem for path expressions in object-oriented database systems, and even to some extent the join optimization problem in relational systems. As will be seen, many of our basic techniques are adapted from prior work in those areas. However, we decided to build a new overall optimization framework for a number of reasons:

- Previous work has considered the optimization of single path expressions (e.g., [GGT96, SMY90]). Our query language permits several, possibly interrelated, path expressions in a single query. We have developed an optimization framework that operates on a set of path expressions, as well as path expressions in the context of a complete query with other query constructs.
- The statistics maintained by relational DBMS's (for joins) and object-oriented DBMS's (for path expression evaluation) are generally based on single joining pairs or object references, while for accuracy in our environment it is essential to maintain more detailed statistics about complete paths.
- The capabilities of deployed OODBMS optimizers are fairly limited, and we know of no available prototype optimizer flexible enough to meet our needs. Building our own framework has allowed us to experiment with and identify good search strategies and pruning heuristics for our large plan space. It also has allowed us to integrate the optimizer easily and completely into the existing Lore system.

2 Related Work

Lore. The first version of the Lorel query language and initial ideas for the Lore system were presented in [QRS+95]. Details of the syntax and semantics of the current version of Lorel can be found in [AQM+97]. The overall architecture of the Lore system, including the simple query processing strategy we used prior to developing our cost-based query optimizer, can be found in [MAG+97].

Other semistructured databases. The *UnQL* query language [BDHS96, FS98] is based on a graph-structured data model similar to OEM. For query optimization, a translation from UnQL to *UnCAL* is defined [BDHS96], which provides a formal basis for deriving optimization rewrite rules such as pushing selections down. However, UnQL does not have a cost-based optimizer as far as we know. The *Strudel* Web-site management system is based on semistructured data [FFLS97, FFK^+99], and query optimization is considered in [FLS98]. In Strudel, semistructured data graphs are introduced for modeling and querying, while the data itself may reside elsewhere in arbitrary format. A key feature of Strudel's approach to query optimization is the use of declarative *storage descriptors*, which describe the underlying data stores. The optimizer enumerates query execution plans, with a cost model that derives the costs of operators from their descriptors. In contrast, Lore data is stored under our control, and the user may dynamically create indexes to provide efficient access methods depending upon the expected queries. Finally, [FLS98] includes detailed experimental results of how large their search space is, but no other performance data is given. In contrast, our experiments focus on the performance of the query plan selected by the optimizer versus other possible query plans.

Some much earlier systems, such as *Multos* [BRG88] and *Model 204* [O'N87], considered problems associated with semistructured data but in very different settings. Multos operated on *complex data objects* which allowed, among other things, sets and pointers to objects of any type. Basic knowledge of the schema was crucial, however, and queries were placed into categories with a fixed set of execution strategies for each category. Lore follows a more traditional and flexible model of query processing. Model 204 was based on self-describing record structures somewhat resembling OEM, but the work concentrated primarily on clever bit-mapped indexing structures to achieve high performance for its relatively simple queries.

Relational databases. As mentioned earlier, at a coarse level the problem of optimizing a Lorel path expression is similar to the join ordering problem in relational databases. However, join ordering algorithms usually rely on statistics about each joining pair, while for typical queries in our environment it is crucial to maintain more comprehensive statistics about entire paths. The computation and storage of our statistics is further complicated by the lack of a schema. In addition, when quantification is present in our queries, the SQL translation results in complex subqueries that many relational optimization frameworks are ill-equipped to handle.

Object-oriented databases. Many of the points discussed in the previous paragraph apply to object-oriented databases as well. There has been some work on optimizing path expressions in an OODBMS context [GGT96]. They propose a set of algorithms to search for objects satisfying path expressions containing predicates, and analyze their relative performance. Our work differs in that we consider many interrelated path expressions within the context of an arbitrary query with other language constructs. We also provide additional access methods for path expressions, and our optimization techniques are implemented within a complete DBMS. Similar comparisons can be drawn between our work and other recent OODB optimization work, e.g., [GGMR97, KMP93, OMS95, SO95]. Some recent papers have specified cost models for object-oriented DBMSs, e.g., [BF97, GGT95]. Object-oriented databases typically support object clustering and physical extents, rendering many of these formulas inapplicable in our setting.

Generalized path expressions. Other recent work, including [FLS98] discussed above, has considered the problem of optimizing the evaluation of *generalized path expressions*, which describe traversals through data and may contain regular expression operators. In [CCM96] an algebraic framework for the optimization of generalized path expressions in an OODBMS is proposed, including an approach that avoids exponential blow-up in the query optimizer while still offering flexibility in the ordering of operations. In [FS98] two optimization techniques for generalized path expressions are presented, *query pruning* and *query rewriting using state extents*. Lore's techniques for handling generalized path expressions are covered in [MW99a] and are not the focus of this paper.

XML query languages. The *XML-QL* data model and query language [DFF+98] is similar in expressibility to ours, with some extensions specific to the current specification of XML. *XQL* [RLS98] is a simpler query language based on single path expressions and is strictly less powerful than XML-QL, Lorel [AQM+97], StruQL [FFLS97], or UnQL [BDHS96]. To the best of our knowledge no full cost-based query optimizer has been developed for XML-QL or XQL, and the optimization principles presented in this paper should be directly applicable when that task is tackled.

3 Preliminaries

3.1 Data Model

Lore's original data model, *OEM* (for *Object Exchange Model*) [PGMW95], was designed specifically for semistructured data. Data in OEM is schema-less and self-describing, and can be thought of as a labeled directed graph. For example, the very small OEM database in Figure 1 contains (fictitious) information about the Stanford Database Group. The vertices in the graph are *objects*; each object has a unique *object identifier* (oid), such as &5. *Atomic objects* have no outgoing edges and contain a value from one of Lore's basic atomic types such as `integer`, `real`, `string`, `gif`, `java`, `audio`, etc. All other objects may have outgoing edges and are called *complex objects*. Object &3 is complex and its *subobjects* are &8, &9, &10, and &11. Object &7 is atomic and has value "Clark". *Names* are special labels that serve as aliases for single objects and as entry points into the database. In Figure 1, *DBGroup* is a name that denotes object &1.

The correspondence between OEM and XML is evident: OEM's objects correspond to elements in XML, and OEM's subobject relationship mirrors element nesting in XML. The fundamental differences are that subelements in XML are inherently ordered since they are specified textually, and XML elements may optionally include a list of attribute-value pairs. Note that graph structure (multiple incoming edges) must be specified in XML with explicit references, i.e., via *ID* and *IDREF(S)* attributes [BPSM98]. The following XML fragment corresponds to the rightmost `Member` in Figure 1, where `Project` is an attribute of type IDREFS.

```
<Member Project="&5 &6">
   <Name>Jones</Name>
   <Age>46</Age>
   <Office> <Building>Gates</Building> <Room>252</Room> </Office>
</Member>
```

As mentioned earlier, we are migrating Lore to use a native XML data model instead of OEM, but most results in this paper carry over to the new version of Lore with few modifications.

We now introduce two definitions that are useful in the remainder of the paper.

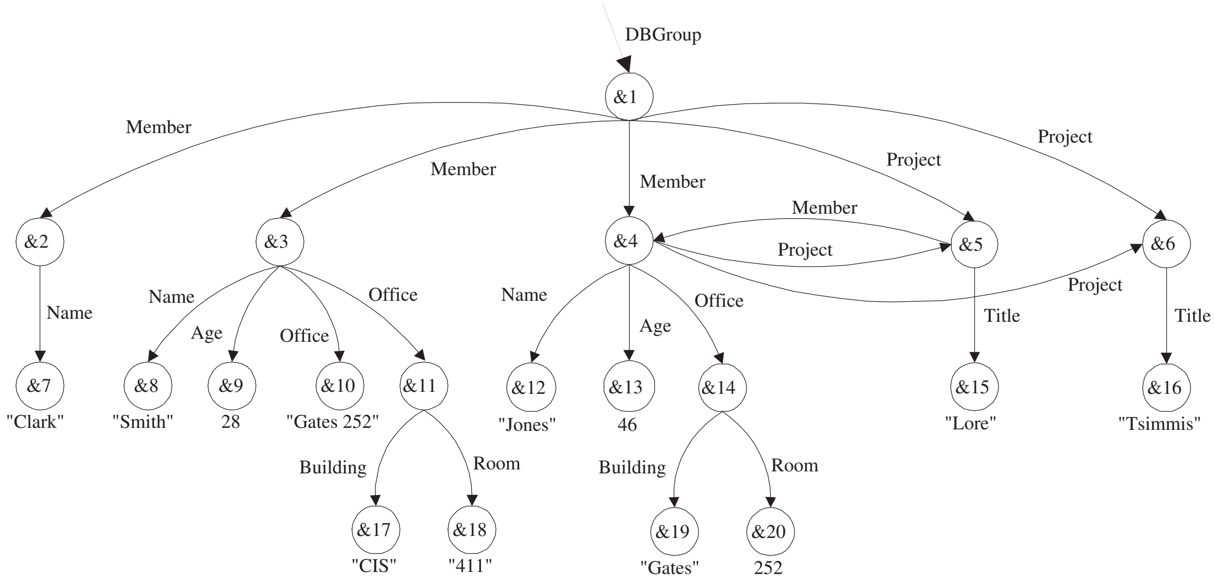

Figure 1: A tiny OEM database

Definition 3.1 (**Simple Path Expression**) A *simple path expression* specifies a single-step navigation in the database. A simple path expression for a variable or name x and label l has the form $x.l\ y$, and denotes that variable y ranges over all l-labeled subobjects of the object assigned to x. If x is an atomic object, or if l is not an outgoing label from x, then y ranges over the empty set.

Definition 3.2 (**Path Expression**) A *path expression* is an ordered list of simple path expressions.

Path expressions are the basic building blocks in the Lorel language and describe traversals through the data in a declarative fashion. For example, "`DBGroup.Member x, x.Age y`" says that variable y ranges over all objects that can be reached by starting with the `DBGroup` object, following an edge labeled `Member`, then following an edge labeled `Age`. Lorel supports a shorthand to write this path expression as "`DBGroup.Member.Age y`", and further shorthands to eliminate variables such as y [AQM+97], however for clarity we avoid shorthands in the examples in this paper. Also, a simple path expression may contain a regular expression or "wildcards" as described in [AQM+97]. In general, l in Definition 3.1 could be a component of a generalized path expression, but we have simplified the definition for presentation purposes in this paper.

3.2 Query Language

Lorel is an extension of OQL [Cat94] supporting declarative path expressions for traversing graph data and extensive automatic coercion for handling heterogeneous and typeless data without generating errors [AQM+97]. Although Lorel offers much syntactic sugar over OQL that is convenient in practice (including the shorthands mentioned above), in this paper we write our queries without these syntactic conveniences in order to be very explicit and enable understanding for those familiar with OQL but unfamiliar with Lorel. As a simple example, consider the following query, which asks for all of the young members of the Database Group.[1] The result of the query over the database of Figure 1 is shown.

```
QUERY 1:                          RESULT: <Member>
  Select x                                  <Name>Smith</Name>
  From   DBGroup.Member x                   <Age>28</Age>
  Where  exists y in x.Age: y<30            <Office>Gates 252</Office>
                                            <Office>
                                               <Building>CIS</Building>
                                               <Room>411</Room>
                                            </Office>
                                          </Member>
```

3.3 Lore Query Processing

The general architecture of the Lore system is very much like a traditional DBMS [MAG+97]. After a query is *parsed*, it is *preprocessed* to factor out common subexpressions and convert Lorel shorthands into a more traditional OQL form. The *logical query plan generator* then creates a single logical query plan describing a very high-level execution strategy for the query. As we will show in Section 5.1, generating logical query plans is fairly straightforward, but special care was taken to ensure that the logical query plans are flexible enough to be transformed easily into vastly different physical query plans. The "meat" of the query optimizer occurs in the *physical query plan enumerator*. This component uses statistics and a cost model in order to transform the logical query plan into the estimated best physical plan that lies within our search space. The physical query plan is a tree composed of *physical operators* that are implemented by the *query execution engine* and perform the low-level steps required to execute the query and construct the result. We use a recursive *iterator* approach in query processing, as described in, e.g., [Gra93], and we assume the reader is familiar with the basic concepts associated with iterators.

3.4 Lore Indexes

As in a conventional DBMS, indexes in Lore enable fast and efficient access to the data. In a relational DBMS, an index is created on an attribute in order to locate tuples with particular attribute values quickly. In Lore, such a *value index* alone is not sufficient, since often the path to a node is as important as the node's value. Lore contains several indexing structures that are useful for finding relevant atomic values, parents of objects, and specific paths and edges within the database. The *value index*, or Vindex, supports finding all atomic objects with a given incoming edge label and satisfying a given predicate. The *label index*, or Lindex, supports finding all parents of a given object via an edge with a given label. The *edge index*, which we term the Bindex, supports finding all parent-child object pairs connected via a specified label. In addition to these indexes, Lore's *DataGuide* [GW97] provides the functionality of a path index, or Pindex. Details on Lore indexes, including coercion issues addressed by the Vindex, can be found in [MWA+98].

[1] The existential quantification in the `Where` clause is necessary since a `Member` object could conceivably have many `Age` subobjects. A shorthand in Lorel allows simply "`where x.Age < 30`", which is preprocessed automatically into the query as shown here [AQM+97].

4 Motivation

As in any declarative query language, there are many ways to execute a single Lorel query. Let us consider Query 1 introduced in Section 3.2 and roughly sketch several types of query plans. As we will illustrate, the optimal query plan depends not only on the values in the database but also on the *shape* of the graph containing the data. It is this additional factor that makes optimization of queries over XML data both important and difficult.

The most straightforward approach to executing Query 1 is to fully explore all `Member` subobjects of `DBGroup` and for each one look for the existence of an `Age` subobject of the `Member` object whose value is less than 30. We call this a *top-down* execution strategy since we begin at the named object `DBGroup` (the top), then process each simple path expression in a forward manner. This approach is similar to *pointer-chasing* in object-oriented systems, and to nested-loop index joins in relational systems. This query execution strategy results in a depth-first traversal of the graph following edges that appear in the query's path expressions.

Another way to execute Query 1 is to first identify all objects that satisfy the "`y < 30`" predicate by using an appropriate Vindex if it exists (recall Section 3.4). Once we have an object satisfying the predicate, we traverse backwards through the data, going from child to parent, matching path expressions in reverse using the Lindex. We call this query execution strategy *bottom-up* since we first identify atomic objects and then attempt to work back up to a named object. This approach is similar to *reverse pointer-chasing* in object-oriented systems. The advantage of this approach is that we start with objects guaranteed to satisfy the `Where` predicate, and do not needlessly explore paths through the data only to find that the final value does not satisfy the predicate. Bottom-up is not always better than top-down, however, since there could be very few paths satisfying the path expression but many objects satisfying the predicate.

A third strategy is to evaluate some, but not all, of a path expression top-down and create a temporary result of satisfying objects. Then use the Vindex as described earlier and traverse up, via the Lindex, to the same point as the top-down exploration. A join between the two temporary results yields complete satisfying paths. (In fact certain join types do not require temporary results at all.) We call this approach a *hybrid* plan, since it operates both top-down and bottom-up, meeting in the middle of a path expression. This type of plan can be optimal when the fan-in degree of the reverse evaluation of a path expression becomes very large at about the same time that the fan-out degree in the forward evaluation of the path expression becomes very large.

These three approaches give a flavor of the very different types of plans that could be used to evaluate a simple query, one that effectively consists of a single path expression. The actual search space of plans for this simple query is much larger, as we will illustrate in Section 5.4, and more complicated queries with multiple interrelated path expressions naturally have an even larger variety of candidate plans.

To make things even more concrete, suppose we are processing the query "`Select x From A.B x Where exists y in x.C: y = 5`", which is isomorphic to Query 1. In Figure 2 we present the general shape and a few atomic values for three databases, illustrating cases when each type of query plan described above would be a good strategy. The database on the left has only one `A.B.C` path and top-down execution would explore only this path. Bottom-up execution, however, would visit all the leaf objects with value 5, and their parents. The second database has many `A.B.C` paths, but only a single leaf satisfying the predicate, so bottom-up is a good candidate. Finally, in

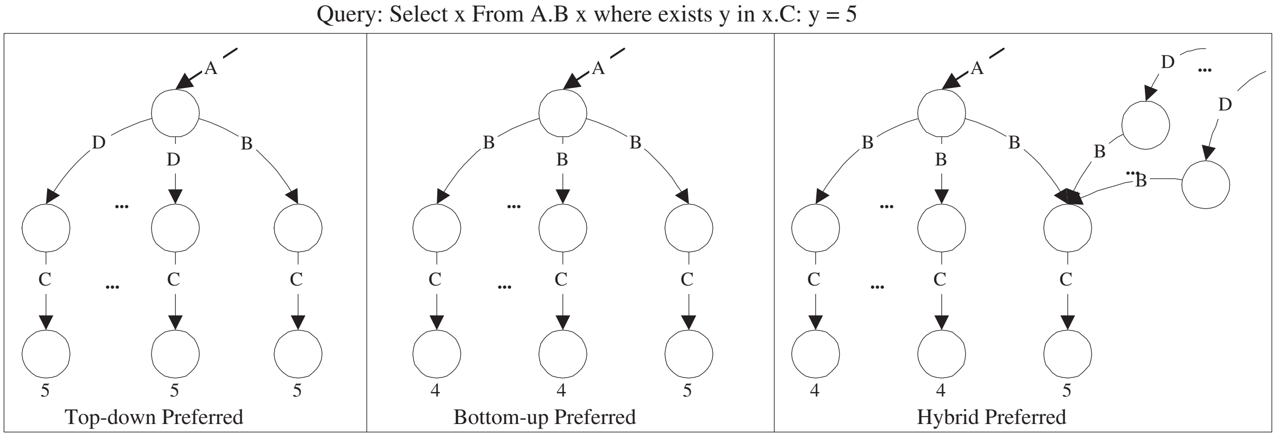

Figure 2: Different databases and good query execution strategies

the third database top-down execution would visit all the leaf nodes, but only a single one satisfies the predicate. Bottom-up would identify the single object satisfying the predicate, but would visit all of the nodes in the upper-right portion of the database. For this database, a hybrid plan where we use top-down execution to find all `A.B` objects, then bottom-up execution for one level, then finally join the two results, would be a good strategy.

Each of these three example plans has a substantially different shape from the others, and each is the optimal plan for a particular database. Our primary goal when designing our logical query plans was to create a structure that represents, at a high level, the sequence of steps necessary to execute a query, while at the same time permits simple rules to transform the logical query plan into a wide variety of different physical query plans.

5 Query Execution Engine

5.1 Logical Query Plans

Recall that a single logical query plan is created after the query is preprocessed into a canonical form. Before explaining the logical query plan operators and structure of the plans, we introduce two additional definitions.

Definition 5.1 (**Variable Binding**) During query processing, a variable x in the query is said to be *bound* if an object o has been assigned to x. We also say that *o is bound to x*.

Definition 5.2 (**Evaluation**) During query processing, an *evaluation* of a query plan (or subplan) is a list of all variables appearing in the plan along with the object (if any) bound to each variable.

The goal of query execution is to iteratively generate complete evaluations for all variables in the query, producing the set of query results based on these evaluations.

One major difference between the top-down and bottom-up query execution strategies introduced in Section 4 is the order in which the query is processed. In the top-down approach we handle the `From` clause before the `Where`; the order is reversed for the bottom-up strategy. Also

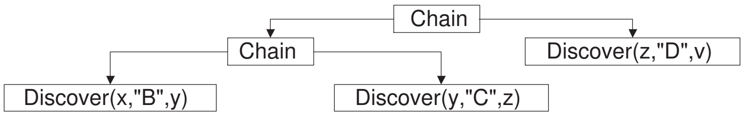

Figure 3: Representation of a path expression in the logical query plan

consider the `Where` clause of Query 1: "`Where exists y in x.Age:y < 30`". We can break this clause into two distinct pieces: (a) find all `Age` subobjects of `x`, and (b) test their values. In the bottom-up plan we first use the Vindex to satisfy (b) and then we use the Lindex for (a). In the top-down strategy first we satisfy (a) by finding an `Age` subobject of x, then we test the condition to fulfill (b).

In fact, all queries can be broken into independent components where the execution order of the components is not fixed in advance. We term the places where independent components meet *rotation points*, since during the creation of the physical query plan we can rotate the order between two independent components to get vastly different physical plans.

In our approach, each logical operator can construct its optimal physical (sub)plan with respect to a set of variables that are already bound elsewhere in the plan. The mechanism by which we store the binding information must also store information about *how* a variable was bound in order for the statistics to accurately estimate the cost and number of results. For example, given the path expression "`x.B y, y.C z`" and assuming that we are following subobjects from variable y to variable z, then the statistics for z will depend on how y was bound. If y was bound via a Bindex operator then the number of object bindings for y (and the number of outgoing `C` edges) might be quite different from the case where y was bound by following subobjects from x. Statistics are discussed further in Section 5.3.

A list and description of most of our logical operators is given in [MW99b]. Here we will focus on the *Discover* and *Chain* logical operators used for path expressions. Each simple path expression in the query is represented as a *Discover* node, which indicates that in some fashion information is discovered from the database. When multiple simple path expressions are grouped together into a path expression, we represent the group as a left-deep tree of *Discover* nodes connected via *Chain* nodes. It is the responsibility of the *Chain* operator to optimize the entire path expression represented in its left and right subplans. As an example, consider the path expression "`x.B y, y.C z, z.D v`" (where x is defined elsewhere in the query) which has the logical query subplan shown in Figure 3. The left-most *Discover* node is responsible for choosing the best way to provide bindings for variables `x` and `y`. The *Chain* node directly above it is responsible for evaluating the path expression "`x.B y, y.C z`" efficiently. This could be done by using the children's most efficient ways of executing their subplans and joining them together in some fashion, or possibly by using a path index for the entire path expression. The final *Chain* and *Discover* nodes are similar.

Figure 4 shows the complete logical query plan for Query 1. Each rotation point is represented by a *Glue* node that has as its children the two independent subplans. The topmost *Glue* node connects the subplans for the `From` and `Where` clauses. The *Chain* node connects the two components of the path expression appearing in the `From`. The *Exists* node quantifies y. A *Glue* node separates the existential in the `Where` from the actual predicate test, allowing either operation to occur first in the physical query plan. Because the semantics of Lorel requires a set of objects to be returned,

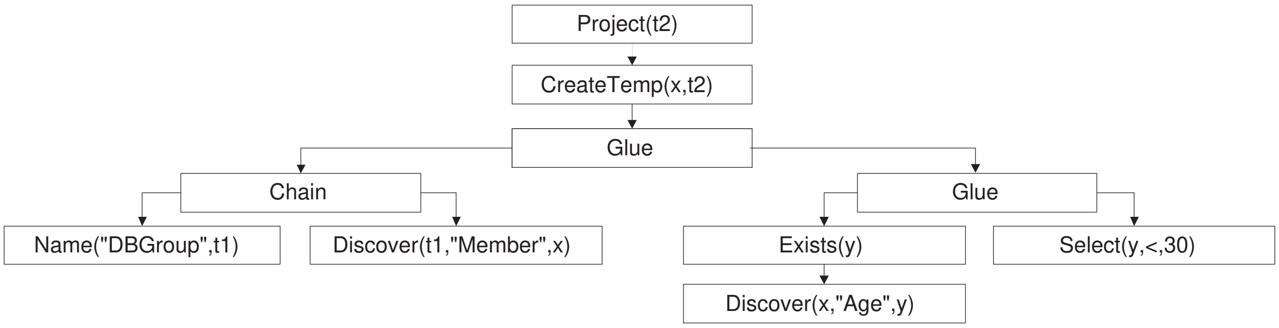

Figure 4: A complete logical query plan

the *CreateTemp* and *Project* nodes at the top of the plan are responsible for gathering the satisfying evaluations and returning the appropriate objects to the user.

Space limitations preclude a full description of logical query plans or examples of more complex queries, but the general flavor and flexibility of our approach should be evident. For details please see [MW99b].

5.2 Physical Query Plans

The number of physical query plan operators is large; a list and description of the more common operators appears in [MW99b]. Here we focus on some of the more interesting operators including those used to traverse paths through the data. Recall that our physical query plan operators are iterators: each node in the plan requests a "tuple" at a time from its children, performs some operation on the tuple(s), and passes result tuples to its parent. The "tuples" that our query plans operate over are evaluations (Definition 5.2): vectors of bindings for variables in the query.

In a physical query plan, there are six operators that identify information stored in the database:

1. *Scan*(x, l, y): The *Scan* operator performs pointer-chasing: it places into y all objects that are subobjects of the complex object x via an edge labeled l.

2. *Lindex*(x, l, y): In the reverse of the *Scan* operator, the *Lindex* operator places into x all objects that are parents of y via an edge labeled l. This reverse pointer-chasing operator is implemented in Lore by the link index (Section 3.4).

3. *Pindex*(*PathExpression*, x): Lore maintains a dynamic "structural summary" of the current database called a *DataGuide* [GW97]. The DataGuide also can be used as a *path index*, enabling quick retrieval of oid's for all objects reachable via a given path expression. The *Pindex* operator places into x all objects reachable via the *PathExpression*.

4. *Bindex*(l, x, y): The *Bindex* operator finds all parent-child pairs connected via an edge labeled l. This operator allows us to efficiently locate edges whose label appears infrequently in the database.

5. *Name*(x, n): The *Name* operator simply verifies that the object in variable x is the named object n.

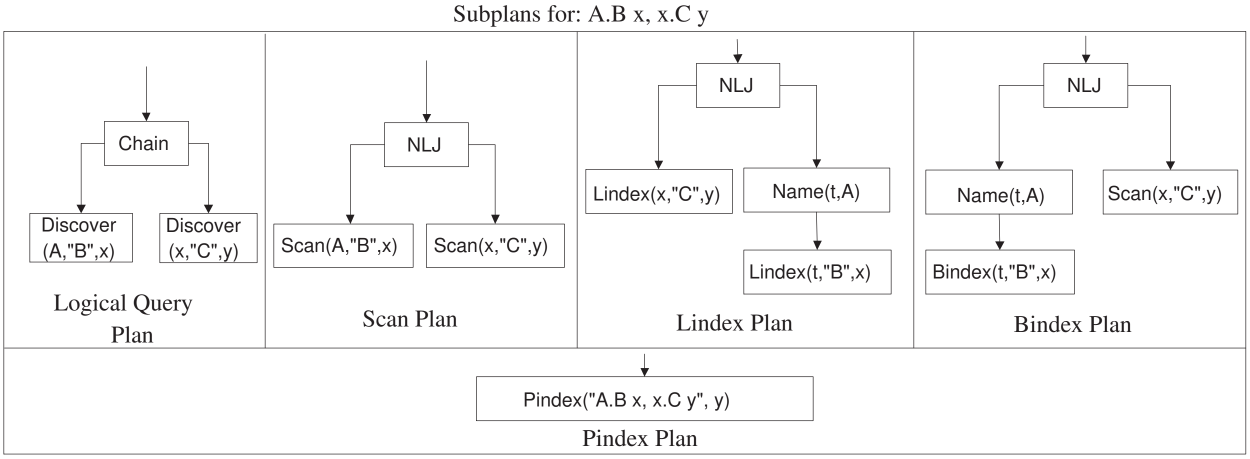

Figure 5: Different physical query plans

6. *Vindex*(*Op*, *Value*, l, x): The *Vindex* operator accepts a label l, an operator *Op*, and a *Value*, and places into x all atomic objects that satisfy the "*Op Value*" condition and have an incoming edge labeled l.

As an example that uses some of these operators, consider the path expression "`A.B x, x.C y`" (where `A` is a name) and four possible plans as shown in Figure 5. (The optimizer can generate up to eleven different physical plans for this single path expression.) The logical query plan is shown in the top left panel. In the first physical plan, the "Scan Plan", we use a sequence of *Scan* operators to discover bindings for each of the variables, which corresponds to the top-down execution strategy introduced in Section 4. If we already have a binding for y then we can use the second plan, the "Lindex Plan". In this plan we use two *Lindex* operations starting from the bound variable y, and then confirm that we have reached the named object `A`. This corresponds to the bottom-up execution strategy of Section 4. In the "Bindex Plan", we directly locate all parent-child pairs connected via a `B` edge using the *Bindex* operator. We then confirm that the parent object is the named object `A`, and *Scan* for all of the `C` subobjects of the child object. In the "Pindex Plan", we use the *Pindex* operator, which allows us to directly obtain the set of objects reached via the given path expression. Note that several of the plans use a nested-loop join (NLJ) operator without a predicate. These are dependent joins where the left subplan passes bound variables to the right subplan.

Recall the hybrid query execution strategy introduced in Section 4. One subplan evaluates a portion of the query and obtains bindings for a set of variables, say V, and another subplan obtains bindings for another set of variables, say W. Suppose $V \cap W$ contains one variable, but the plans are otherwise *independent*, meaning one does not provide a binding that the other uses (as in the hybrid plan). Then by creating evaluations for both subplans and joining the results on the shared variable, we efficiently obtain complete evaluations. As in relational systems, deciding which join operator to use is an important decision made by the optimizer. (Currently we support nested-loop and hash joins.)

Again space limitations preclude a full description of physical query plans or examples of even remotely complicated queries, although we will visit physical query plans for Query 1 later in

Section 5.4. In addition to the basic traversal operators discussed above, we have operators to perform projection and selection, manage temporary results, perform an aggregation operation over a subplan, ensure the existential and universal quantification of a variable, perform set and arithmetic operations between two subplans, and others. For details please see [MW99b]. The physical operators can be combined in numerous ways, producing a vast search space for even relatively simple queries. Our plan enumeration and pruning heuristics will be discussed in Section 5.4.

5.3 Statistics and Cost Model

As with any cost-based query optimizer, we need to establish a metric by which we estimate the execution cost for a given physical query plan or subplan. Lore currently does not enforce any object clustering, so we are limited to using the predicted number of object fetches as our measure of I/O cost, since we cannot accurately determine whether two objects will be on the same page. Despite this rough approximation, experiments presented in Section 6 validate that our cost model is reasonably accurate. Nevertheless, refining and expanding the cost model, especially by increasing our knowledge of the locality of objects on pages (through statistics-gathering and/or actual clustering), is an area where we intend to invest future effort.

5.3.1 Statistics

Our query optimizer must consult statistical information about the size, shape, and range of values within a database in order to estimate the cost of a physical query plan. Initially we stored statistics in the *DataGuide*, but quickly were limited by the fact that we could only store statistics about paths beginning from a named object [GW97]. The optimizer may choose to begin evaluating a path expression anywhere within the path (via the *Bindex* or *Vindex* operator), so we needed more flexible statistics. Our new approach is to store statistics about all possible subpaths (label sequences) in the database up to a length k, where k is a tuning parameter. (Typical object-oriented and relational database systems compute and store statistics for $k = 1$.) We have explored several algorithms for efficiently computing and storing such statistics, but a presentation of these algorithms is outside of the scope of this paper. Regardless of algorithm used, a clear trade-off exists between the cost in time and space for a larger k and the accuracy of the statistics.

The statistics we maintain for every label subpath p of length $\leq k$ include:

- For each atomic type, the total number of atomic objects of that type reachable via p.
- For each atomic type, the minimum and maximum values of all atomic objects of that type reachable via p.
- The total number of instances of path p, denoted $|p|$.
- The total number of distinct objects reachable via p, denoted $|p|_d$.
- The total number of l-labeled subobjects of all objects reachable via p, denoted $|p_l|$.
- The total number of incoming l-labeled edges to any instance of p, denoted $|p^l|$.

As mentioned earlier, our I/O cost metric is based on the estimated number of objects fetched during evaluation of the query. Thus, for example, given an evaluation that corresponds to a traversal to some point in the data, the optimizer must estimate how many objects will bind to the next simple path expression to be evaluated. Consider evaluating the path expression "`A.B x, x.C y`" top-down. If we have a binding for x, then the optimizer needs to estimate the number of `C` subobjects, on average, that objects reached by the path "`A.B x`" have. Alternatively, if we proceed bottom-up with a binding for x, then the optimizer must estimate the average number of parents via a `B` edge for all the `C`'s. We call these two estimates *fan-out* and *fan-in* respectively. The fan-out for a given path expression p and label l is computed from the statistics by $|p| * (|p_l|/|p|_d)$. Likewise, fan-in is $|p| * (|p^l|/|p|_d)$.

Our statistics are most accurate for path expressions of length $\leq k+1$, since for a given k we store statistics about paths of length up to k, and these statistics include information about incoming and outgoing edges to the paths—effectively giving us information about all paths of length $k+1$. Given a path expression of length $k+2$, for maximum accuracy we combine the statistics for two overlapping paths p_1 and p_2 each of length $k+1$.

When estimating the number of atomic values that will satisfy a given predicate, standard formulas such as those given in [SAC+79, PSC84] are insufficient in our semistructured environment due to the extensive type coercion that Lore performs [AQM+97]. Our formulas take coercion into account by combining value distributions for all atomic types that can be coerced into a type comparable to the value in the predicate.

5.3.2 Cost Model

Each physical query plan is assigned a *cost* based upon the estimated I/O and CPU time required to execute the plan. The costing procedure is recursive: the cost assigned to a node in the query plan depends on the costs assigned to its subplans, along with the cost for executing the node itself. In order to compute estimated cost recursively, at each node we must also estimate the number of evaluations expected for that subplan. To decide if one plan is cheaper than another, we first check the estimated I/O cost. Only when the I/O costs are identical do we take estimated CPU cost into account. Again, our cost metric is admittedly simplistic, but it does appear acceptable for the first version of our cost-based optimizer as shown by the performance results in Section 6.

Due to space constraints, our formulas for estimated I/O and CPU cost and number of evaluations are omitted but appear in [MW99b]. As an example, consider the I/O formula for the *Vindex*(*Op*, *Value*, l, x) operator: $BLevel_{l,type1}+$ $Selectivity_1(l,\ Op,\ Value)$ $+BLevel_{l,type2}+$ $Selectivity_2(l,\ Op,\ Value)$. Here *BLevel* gives the height of the relevant B+-tree index, and the *Selectivity* functions are the formulas to estimate the number of satisfying results given Lore's coercion system. (Because of type coercion, multiple B+-trees need to be accessed during a *Vindex* operation.) As a second example, the I/O cost for the *Lindex*(x, l, y) operator is $2 + Fin_{PathOf(y),l}$, where *Fin* is the fan-in statistic as defined earlier. The *Lindex* is implemented using *extendible hashing* [FNPS79], and our cost estimate assumes no overflow buckets. Thus, it requires two page fetches (one for the directory and one for the hash bucket) and one additional page fetch for every possible parent.

n: number of simple path expressions	n=3	n=5	n=7
All possible plans/Lore's search space	1458 / 48	2,361,960 / 228	8,035,387,920 / 948

Table 1: Analysis of Search Space Size

5.4 Plan Enumeration

The search space of physical query plans for a single Lorel query is very large. For example, a single path expression of length n can be viewed as an n-way join, where as "join methods" Lore considers pointer-chasing, reverse pointer-chasing, and two different standard relational joins. Furthermore, there may be many interrelated path expressions in a single query, along with other constructs such as set operations, subqueries, aggregation, etc. [AQM+97]. In order to reduce the search space as well as the complexity of our plan enumerator, we use a greedy approach to generating physical query plans. Each logical query plan node makes a locally optimal decision, creating the best physical subplan for the logical plan rooted at that node. The decision is based on a given set of bound variables passed from the node's parent. The key to considering a variety of different physical plans is that a node may ask its child(ren) for the optimal physical query subplan many times, using different sets of bound variables each time. While this greedy approach greatly reduces the search space, it still explores an exponentially large number of physical query plans. Thus, our plan enumerator currently uses the following additional heuristics to further prune the search space.

- The optimizer does not consider joining two simple path expressions together unless they share a common variable. This restriction substantially reduces the number of ways to order the evaluation of simple path expressions. (See [MW99b] for a detailed discussion.)
- The *Pindex* operator is considered only when a path expression begins with a name, and no variable except the last is used elsewhere within the query. The latter requirement is based on the fact that *Pindex* only binds the last variable in its path expression, so other needed variables in the path would have to be discovered by some additional method.
- The `Select` clause always executes last, since in nearly all cases it depends on one or more variables bound in the `From` clause. Also, the physical query plan will always execute either the complete `From` or complete `Where` clause before moving on to the other one.
- The optimizer does not attempt to reorder multiple independent path expressions.

A detailed analysis of our physical query plan search space is provided in [MW99b]. Table 1 gives some examples of how dramatically our heuristics reduce the search space. However, even with our aggressive pruning, Lore still chooses very good plans as we demonstrate in Section 6. For further refinement, we intend to experiment with a final optimization phase in which we can apply transformations directly over the generated physical query plan, such as moving subplans to different locations within the overall plan.

We now discuss how physical plans are produced. As mentioned earlier, each logical plan node creates an optimal physical plan given a set of bound variables. During plan enumeration we track for every variable in the query: (1) whether the variable is bound or not; (2) which plan operator

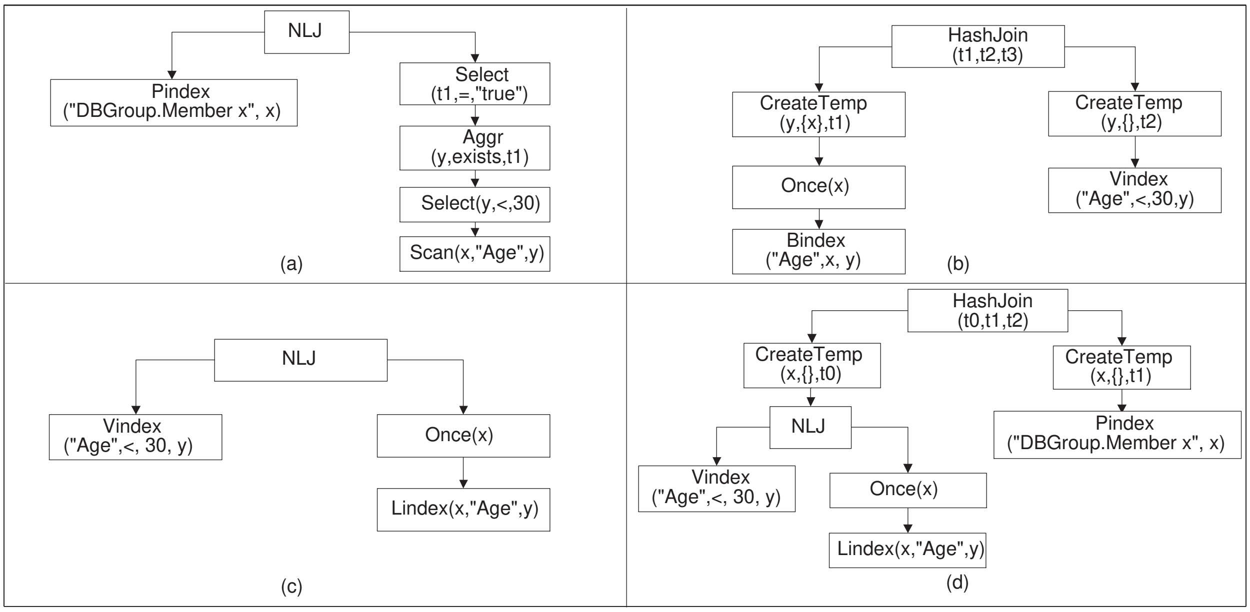

Figure 6: Possible transformations for Query 1 into a physical query plan

has bound the variable; (3) all other plan operators that use the variable; (4) whether the variable is stored within a temporary result. For instance, the logical query plan node *Discover* for the simple path expression `x.Age y` may be asked to create its optimal plan given that x has already been bound by some other physical operator, in which case it may decide that *Scan* is optimal. However, if y was bound instead then it may decide that *Lindex* is optimal. After a node creates its optimal subplan, the new state of the variables and the optimal subplan are passed to the parent. Note that a logical node may be unable to create any physical plan for a given state of the variables if it always requires some variables to be bound. In this case, "no plan" is returned and a different choice must be made at a higher level in the plan. In [MW99b] we detail how each logical plan node generates its optimal physical subplan.

To illustrate the transformation from a logical plan to a physical plan, let us consider part of the search space explored during the creation of the physical query plan for Query 1, whose logical query plan was given in Figure 4. The topmost *Glue* node (indicating a rotation point) in Figure 4 is responsible for deciding the execution order of its children: either left-then-right or right-then-left. It requests the best physical query plan from the left child and then, using the returned bindings, requests the best physical query plan from the right child. One possible outcome is the physical query plan fragment shown in Figure 6(a). After exploring left-then-right execution order, the topmost *Glue* node considers the right-then-left order. The right child is another *Glue* node which recursively follows the same procedure. Suppose that for this nested *Glue* node, the left-then-right execution order results in the physical subplan shown in Figure 6(b), while the right-then-left execution order results in Figure 6(c). (For details of the *CreateTemp* and *Once* operators please see [MW99b].) Suppose plan (c) is chosen based on a lower estimated cost. The bindings provided by this subplan are then supplied to the left child of the topmost *Glue* node to create the optimal query plan for the left child, which could result in the final subplan shown in Figure 6(d). Notice that in the right subplan for the topmost *Glue* node, the *Chain* node decided that the *Pindex* operator is the best way to get all "`DBGroup.Member x`" objects within the database, despite the fact that we already have a binding for x. This choice makes sense when the estimated fan-in for x with label `DBGroup` is very high. As a final step the topmost *Glue* node decides which query plan is cheaper, either (a) or (d), and passes that plan to its parent.

6 Performance Results

The query optimization techniques described in this paper are fully implemented and integrated into Lore, including the physical operators, statistics, cost formulas, logical query plan generation, and physical query plan enumeration and selection. The implementation for these components consists of approximately 31,000 lines of C++ code. We also have implemented mechanisms for instructing the optimizer to favor certain types of plans (in order to debug and conduct experiments), and we have built a very useful query plan visualization tool. We now present some preliminary performance results showing that our cost model is reasonably accurate and that the optimizer is choosing good plans. Extensive performance evaluations over a large suite of queries and databases is beyond the scope of this paper.

All of our tests were run on a Sun Ultra 2 with 256 megabytes of RAM. However, Lore was configured to have a small buffer size of approximately 150K bytes, in order to match the relatively small databases used by our initial performance experiments. Each query was run with an initially empty buffer. Over all of the queries in our experiments the average optimization time was approximately 1/2 second.

At the time of this writing we have not located any significant amounts of readily available XML data. What is available consists mostly of small, tree-structured documents usually with cryptic tags or presentational tags borrowed from HTML. Rather than use these small datasets for our experiments we built our own XML database about movies made in 1997, combining information from many sources including the Internet Movie Database (located at `http://www.imdb.com`). The database contains facts about 1,970 movies, 10,260 actors and actresses, plot summaries, directors, editors, writers, etc., as well as multimedia data such as stills and audio clips. The database loaded into Lore is about 5 megabytes. Value, Link, and Edge indexes (recall Section 3.4) account for an additional 8.1 megabytes. The database is semistructured and very cyclic; for example, actors have edges to each movie they appeared in (along with their role in the movie), and movies have edges to all of the actors in the movie. The database graph contains 62,256 nodes and 130,402 edges.

Lore allows us to turn off all pruning heuristics temporarily, in order to create and execute all possible query execution plans within our search space for a single query. Thus, we can evaluate how the chosen plan performs against other possible plans. However, it is infeasible to perform this extensive experiment for large queries, since the number of plans in the search space is very large, and some plans are extremely slow to execute (even if the chosen plan is very efficient). We report on a sample of experiments, again emphasizing that exhaustive performance evaluations are beyond the scope of this paper.

Experiment 1. We begin with an extremely simple query: `Select DB.Movie.Title`. Using exhaustive search Lore produces eleven different query plans, with estimated I/O costs and actual execution times (in seconds) as follows:

Plan #	1*	2	3	4	5	6	7	8	9	10	11
Exec. Time (sec.)	0.36	1.78	2.02	14.44	61.82	67.24	74.09	94.15	250.61	397.18	423.34
Estimated I/O	1975	3944	3944	9853	31918	31918	11823	37827	17742	17733	23855

In this and subsequent tables the plan chosen by the optimizer is marked with a star (*). The first and best plan simply uses Lore's path index to quickly locate all the movie titles. The second plan, which is only slightly slower, uses top-down pointer-chasing. The worst plan uses *Bindex* operators and hash joins.

To evaluate the relative accuracy of our cost model, consider the estimated I/O cost against the actual execution time. With some exceptions, the estimated cost accurately reflects the relative execution time for each plan. Since our cost model is still quite simplistic, we are very encouraged by these results. □

Experiment 2. Our second query asks for all movies with a `Genre` subobject having value "Comedy". This turned out to be a "point" query, since many movies don't have a `Genre` subobject and most aren't comedies. Estimated I/O costs again reflected relative execution times fairly accurately, so hereafter we focus only on execution times. Twenty-four plans were considered using exhaustive search. The following table describes some of them, where "plan rank" indicates rank by execution time among all plans considered.

Plan Rank	Execution Time	Description
1*	0.3307	Bottom-up
2	0.3768	*Bindex* for `Genre` with *Select* then *Lindex*
7	3.3384	Top-down
24	458.58	*Bindex* with hash joins

Since the `Where` clause is very selective, the optimal plan uses a bottom-up strategy: a *Vindex* operator locates all objects having the value "Comedy" and an incoming edge labeled `Genre`. The *Lindex* operator matches the remainder of the path expression in reverse. The second-best plan is only slightly slower. It uses the *Bindex* to locate all `Genre` edges, filters using the "Comedy" predicate, then proceeds bottom-up. The slowest plan uses a poor combination of *Bindex* operators and joins. Top-down evaluation results in the seventh-best plan. □

Experiment 3. In the remaining two experiments we did not execute all possible plans since the queries and space of plans are much larger. Instead, we generated different plans by forcing or disallowing the use of particular operators or indexes. The following results are for a query with two existentially quantified variables in the `Where` clause.

Plan Rank	Execution Time	Description
1	0.33	Bottom-up
2*	3.68	Top-down
3	6.95	Hybrid with *Pindex*
4	7.01	Hybrid with pointer-chasing
5	23.13	*Bindex* over Movie with *Vindex* then *Lindex*

Notice that the optimizer chose plan 2, the top-down or pointer-chasing execution strategy, as the best plan. The mistake is due largely to simplistic estimates of atomic value distributions (we have not yet implemented histograms) and of set operation costs. Devoting some effort to improving the optimizer in these areas should lead it to select the optimal plan. □

Experiment 4. Our fourth query selects movies with a certain quality rating. Here too we considered only a sampling of all possible plans.

Plan Rank	Execution Time	Description
1*	0.61	*Bindex* for rating, then *Lindex* up
2	0.89	Bottom-up
3	4.04	Top-down

Since it turns out that quality ratings are fairly uncommon in the database, the optimizer (correctly) chooses to find all ratings via the *Bindex*, then to work bottom-up. □

We have performed many experiments in addition to those reported here, although the ones described are a representative sample. In general, our experiments so far allow us to conclude: (i) our cost estimates are accurate enough to select the best plan in most cases, although some refinements are needed; (ii) execution times of the best and worst plans for a given query and database can differ by many orders of magnitude; and (iii) the best execution strategy is highly dependent on the query and database, indicating that a cost-based query optimizer for XML data is crucial to achieving good performance.

7 Future Work

As mentioned in the introduction, we are in the process of migrating Lore to a fully XML-based data model, and we are extending the Lorel query language accordingly. The primary changes to the model are the introduction of ordered subobjects, attribute-value lists, and *reference* edges in addition to normal subobject edges. Corresponding changes are being made to the query language, although in most cases the Lorel queries one uses over an OEM database are identical to those used over a corresponding XML database. (We expect to detail Lore's XML conversion in a forthcoming short report; please visit our Web site at `www-db.stanford.edu/lore`.) Likewise, changes to the query processor—and in particular the query optimizer as described in this paper—so far are fairly minimal, even when incorporating new forms of equality (to handle XML free-text) and new operators such as accessing subobjects based on their ordinal position.

In addition to the XML conversion, improvements are underway in query optimization, including specific optimization techniques for *branching path expressions*, a query rewrite that moves `Where` clause predicates into the `From` clause, and a transformation that introduces a `Group-By` clause when a large number of paths pass through a small number of objects. We are also considering *partially correlated subplans*, which are similar to correlated subqueries but rely on the bindings passed between portions of the physical query plan rather than on the query itself. In the area of statistics we are considering even more efficient statistics-gathering algorithms, perhaps incorporating some graph sampling. We also plan to gather statistics about the location of objects on disk, with corresponding modifications to the cost formulas to generate more accurate cost estimates.

References

[Abi97] S. Abiteboul. Querying semistructured data. In *Proceedings of the International Conference on Database Theory*, pages 1–18, Delphi, Greece, January 1997.

[AQM+97] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The Lorel query language for semistructured data. *Journal of Digital Libraries*, 1(1):68–88, April 1997.

[BDHS96] P. Buneman, S. Davidson, G. Hillebrand, and D. Suciu. A query language and optimization techniques for unstructured data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 505–516, Montreal, Canada, June 1996.

[BF97] E. Bertino and P. Foscoli. On modeling cost functions for object-oriented databases. *IEEE Transactions on Knowledge and Data Engineering*, 9(3):500–508, May 1997.

[BPSM98] T. Bray, J. Paoli, and C. Sperberg-McQueen. Extensible markup language (XML) 1.0, February 1998. W3C Recommendation available at http://www.w3.org/TR/1998/REC-xml-19980210.

[BRG88] E. Bertino, F. Rabitti, and S. Gibbs. Query processing in a multimedia document system. *ACM Transactions on Office Information Systems*, 6(1):1–41, January 1988.

[Bun97] P. Buneman. Semistructured data. In *Proceedings of the Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, Tucson, Arizona, May 1997. Tutorial.

[Cat94] R.G.G. Cattell. *The Object Database Standard: ODMG-93*. Morgan Kaufmann, San Francisco, California, 1994.

[CCM96] V. Christophides, S. Cluet, and G. Moerkotte. Evaluating queries with generalized path expressions. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 413–422, Montreal, Canada, June 1996.

[DFF+98] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. XML-QL: A query language for XML, August 1998. Available at http://www.w3.org/TR/NOTE-xml-ql/.

[FFK+99] M. Fernandez, D. Florescu, J. Kang, A. Levy, and D. Suciu. Catching the boat with Strudel: Experiences with a web-site management system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 414–425, Seattle, Washington, June 1999.

[FFLS97] M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A query language for a web-site management system. *SIGMOD Record*, 26(3):4–11, September 1997.

[FLS98] D. Florescu, A. Levy, and D. Suciu. Query optimization algorithm for semistructured data. Technical report, AT&T Laboratories, June 1998.

[FNPS79] R. Fagin, J. Nievergelt, N. Pippenger, and H. Strong. Extendible hashing – A fast access method for dynamic files. *ACM Transactions on Database Systems*, 4(3):315–344, September 1979.

[FS98] M. Fernandez and D. Suciu. Optimizing regular path expressions using graph schemas. In *Proceedings of the Fourteenth International Conference on Data Engineering*, pages 14–23, Orlando, Florida, February 1998.

[GGMR97] J. Grant, J. Gryz, J. Minker, and L. Raschid. Semantic query optimization for object databases. In *Proceedings of the Thirteenth International Conference on Data Engineering*, pages 444–454, Birmingham, UK, April 1997.

[GGT95] G. Gardarin, J. Gruser, and Z. Tang. A cost model for clustered object-oriented databases. In *Proceedings of the Twenty-First International Conference on Very Large Data Bases*, pages 323–334, Zurich, Switzerland, September 1995.

[GGT96] G. Gardarin, J. Gruser, and Z. Tang. Cost-based selection of path expression processing algorithms in object-oriented databases. In *Proceedings of the Twenty-Second International Conference on Very Large Data Bases*, pages 390–401, Bombay, India, 1996.

[Gra93] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, 1993.

[GW97] R. Goldman and J. Widom. DataGuides: Enabling query formulation and optimization in semistructured databases. In *Proceedings of the Twenty-Third International Conference on Very Large Data Bases*, pages 436–445, Athens, Greece, August 1997.

[KMP93] A. Kemper, G. Moerkotte, and K. Peithner. A blackboard architecture for query optimization in object bases. In *Proceedings of the Nineteenth International Conference on Very Large Data Bases*, pages 543–554, Dublin, Ireland, August 1993.

[MAG+97] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A database management system for semistructured data. *SIGMOD Record*, 26(3):54–66, September 1997.

[Mar98] M. Marchiori. *Proceedings of QL'98 - The Query Languages Workshop*. Boston, MA, December 1998. Papers available online at http://www.w3.org/TandS/QL/QL98/.

[MW99a] J. McHugh and J. Widom. Compile-time path expansion in Lore. In *Workshop on Query Processing for Semistructured Data and Non-Standard Data Formats*, Jerusalem, Isreal, January 1999.

[MW99b] J. McHugh and J. Widom. Query optimization for semistructured data. Technical report, Stanford University Database Group, February 1999. Document is available as `ftp://db.stanford.edu/pub/papers/qo.ps`.

[MWA+98] J. McHugh, J. Widom, S. Abiteboul, Q. Luo, and A. Rajaraman. Indexing semistructured data. Technical report, Stanford University Database Group, 1998. Document is available as `ftp://db.stanford.edu/pub/papers/semiindexing98.ps`.

[OMS95] M. T. Ozsu, A. Munoz, and D. Szafron. An extensible query optimizer for an objectbase management system. In *Proceedings of the Fourth International Conference on Information and Knowledge Management*, pages 188–196, Baltimore, Maryland, November 1995.

[O'N87] Patrick O'Neil. Model 204 architecture and performance. In *Proceedings of the 2nd International Workshop on High Performance Transaction Systems (HPTS)*, pages 40–59, Asilomar, CA, 1987.

[PGMW95] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *Proceedings of the Eleventh International Conference on Data Engineering*, pages 251–260, Taipei, Taiwan, March 1995.

[PSC84] G. Piatetsky-Shapiro and C. Connell. Accurate estimation of the number of tuples satisfying a condition. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 256–276, Boston, MA, June 1984.

[QRS+95] D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, and J. Widom. Querying semistructured heterogeneous information. In *Proceedings of the Fourth International Conference on Deductive and Object-Oriented Databases*, pages 319–344, Singapore, December 1995.

[RLS98] J. Robie, J. Lapp, and D. Schach. XML query language (XQL). In *[Mar98]*, 1998.

[SAC+79] P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price. Access path selection in a relational database management system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 23–34, Boston, MA, June 1979.

[SMY90] W. Sun, W. Meng, and C. T. Yu. Query optimization in object-oriented database systems. In *Database and Expert Systems Applications*, pages 215–222, Vienna, Austria, August 1990.

[SO95] D. D. Straube and M. T. Ozsu. Query optimization and execution plan generation in object-oriented database systems. *IEEE Transactions on Knowledge and Data Engineering*, 7(2):210–227, April 1995.