Query Optimization for Semistructured Data*

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

Abstract

With the emerging prevalence of *semistructured* data—data that may be irregular or incomplete—it is important to develop efficient query processing techniques for such data. This paper describes the query processor of *Lore*, a DBMS for semistructured data, and focuses particularly on the cost-based query optimization techniques we have developed and implemented for a semistructured environment. While all of the usual problems associated with cost-based query optimization apply to semistructured data as well, a number of additional problems arise, such as vastly different query execution strategies for different semistructured databases, more complicated notions of database statistics, and novel uses of indexing. We introduce very flexible logical query plans that can be transformed into a wide variety of physical plans, define appropriate database statistics and a cost model, and describe plan enumeration including heuristics for reducing the search space. Our optimizer is fully implemented for most of the Lore query language, and preliminary performance results are reported.

1 Introduction

Several recent trends, including the integration of data from heterogeneous sources and the emergence of the World-Wide Web, have sparked a great deal of interest in *semistructured* data [Abi97, Suc97]. Unlike well-structured relational, object-relational, or object-oriented data, semistructured data need not adhere to a fixed schema defined in advance. Rather, the schema may evolve as rapidly as the data does and in unpredictable ways. Furthermore, semistructured data may be irregular or incomplete, and it may exhibit structure and type heterogeneity. Generally it is very difficult to force such data into a traditional strongly-typed data model requiring a predefined schema, so many applications involving semistructured data forgo the use of a database management system, despite the fact that the strengths of a DBMS—ad-hoc queries, efficient access, concurrency control, crash recovery, security, etc.—would be very useful to those applications.

In the *Lore*[1] project at Stanford we are building a database management system designed specifically for semistructured data. We decided to build Lore from scratch so that we could experiment with how the semistructured nature of our data affects each component of the DBMS. For an overview of the Lore system and discussion of the issues we have addressed in its various components see [MAG+97]. Lore has evolved into a full-feature multi-user DBMS and is available to the public; see `www-db.stanford.edu/lore`.

In this paper we address query processing in Lore, and in particular we describe the cost-based query optimizer we recently designed and implemented. (Earlier versions of Lore's query processor

*This work was supported by the Air Force Rome Laboratories and DARPA under Contracts F30602-95-C-0119 and F30602-96-1-031.

[1]The name Lore derives from *Lightweight Object Repository*, initially indicative of the fact that we were building a "lightweight" system for "lightweight" objects [QRS+95].

naively selected from one or two types of simple query plans.) To the best of our knowledge ours is the first complete cost-based optimizer for queries over semistructured data, and because our data model and query language are similar to others [Cat94, BDHS96] our approach should certainly be applicable to other ongoing work in the area.

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 semistructured data complicate the problem. Because we use a graph-based data model (Section 3) and do not have a known fixed schema for a given database, many of our physical operators look very different from those found in a relational or even object-oriented query processor. Furthermore, a given query may have many candidate physical plans with vastly different shapes. A significant challenge for us was to develop logical plans that are flexible enough to generate widely differing physical plans while handling arbitrary queries in our language. Another challenge was to define an appropriate set of database statistics and devise methods for computing and storing them, again without the benefit of a schema. Further, as will be seen, we have developed some novel indexing techniques for query processing in Lore, and much of the plan enumeration phase consists of properly incorporating and evaluating these techniques in query plans. Finally, since our search space is at least as large as one finds in a conventional optimizer, we needed to develop pruning heuristics appropriate to our query plan enumeration strategy.

Section 2 surveys related work. Section 3 introduces the *Object Exchange Model* (*OEM*) [PGMW95], Lore's self-describing graph-based model for semistructured data. Section 3 also briefly describes Lore's query language *Lorel* (for *Lore Language*), mentioning the basic features needed to understand the query optimization process. Details on the full language can be found in [AQM+96]. Section 4 motivates the variety of query execution strategies we can generate through several examples. Section 5 then describes our logical query plans, followed by Section 6 that covers physical plans. Our cost model and statistics are given in Section 7. The plan enumeration algorithm is covered in Section 8, including our overall strategy and our heuristics for pruning the search space. Finally, some preliminary performance results are reported in Section 9, although exhaustive performance evaluation is not the focus of this paper.

2 Related Work

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+96]. 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].

The *UnQL* query language [BDHS96, FS98] is based on a data model similar to ours and also is designed specifically for semistructured data. 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, and no performance results have been reported. A much earlier system, *Model 204* [O'N87], was based on self-describing record structures that can be thought of as semistructured data. Lore's data model is more powerful in that it includes arbitrary object nesting, so Lore's query language is richer than the language of Model 204. Thus, query processing in Lore is more complicated than in Model 204, which concentrated primarily on clever bit-mapped indexing structures to achieve high performance.

Some recent papers have specified cost models for object-oriented DBMS's, e.g., [BF97, GGT95]. Because Lore does not guarantee the object clustering properties assumed by these cost models and does not provide a fixed schema, we were unable to adapt these models immediately for our purposes, and our current cost model is quite a bit more simplistic. As future work we intend to investigate object clustering and whether it would permit us to (among other things) adapt known OODB cost formulas for our setting.

Other recent work has focused on optimizing the evaluation of *generalized path expressions*, which describe traversals through data and may contain regular expressions. In [CCM96] an algebraic framework for the optimization of generalized path expressions is proposed, including an approach that avoids exponential input to the query optimizer and offers 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*. While the Lorel language does include an analogous concept of generalized path expressions within its queries, in this paper we in fact drop that feature (and only that feature) and focus instead on the overall query optimization problem for complete queries. We expect that optimization techniques designed specifically for generalized path expressions, such as those in [CCM96, FS98], can be incorporated into our cost-based query optimizer and we hope to do so in the near future.

In the context of query optimization for standard (well-structured) object-oriented DBMS's, [GGT96] proposes a set of algorithms to search for objects satisfying path expressions containing predicates, and analyzes their relative performance. Our work differs in that we also consider existentially quantified variables, we gather and store statistics in a semistructured environment, we have 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].

3 Preliminaries

We first introduce the context of our query optimization work by describing the data model and query language of Lore, the basic architecture of the Lore query processor, and a description of indexing in Lore. For motivation and further details on the overall Lore system see [MAG$^+$97]. See [AQM$^+$96] for details on Lorel, Lore's query language.

3.1 Data Model

Lore's data model is the *Object Exchange Model* (*OEM*), which is designed for semistructured data [PGMW95]. Data in this model can be thought of as a labeled directed graph. For example, the very small OEM database shown 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

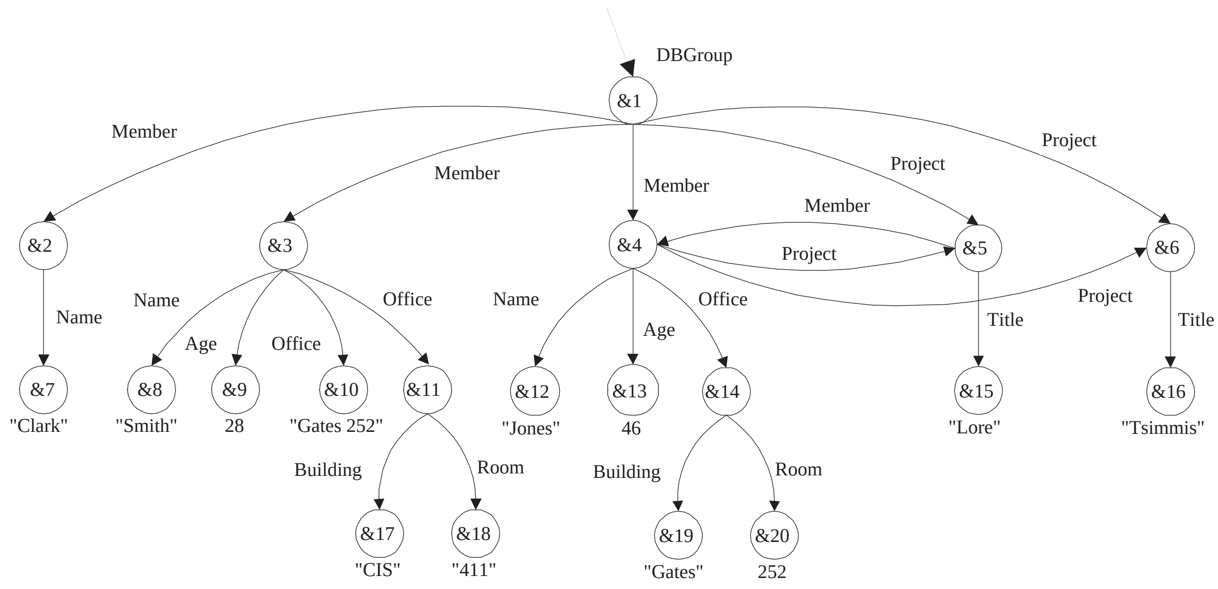

Figure 1: An OEM database

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.

In an OEM database there is no notion of a fixed schema. Instead, all schematic information is included in the labels, which may change dynamically. Thus, an OEM database is *self-describing*, and there is no regularity imposed on the data. The model is designed to handle incompleteness of data, as well as structure and type heterogeneity as exhibited in the example database. Observe in Figure 1 that, for example: (i) members have zero, one, or more offices; (ii) an office is sometimes a string and sometimes a complex object; (iii) a room may be a string or an integer.

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

Definition 3.1 (**Simple Path Expression**) A *simple path expression* specifies a single-step navigation in an OEM database. A simple path expression for an OEM object x and label l has the form $x.l\ y$, and denotes that y ranges over all l-labeled subobjects of 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`" is a path expression. Its semantics say that y ranges over the objects that can be reached 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$^+$96], however for clarity we avoid shorthands in the examples in this paper. Note that this definition of a path expression is more restrictive than Lorel's *general path expressions* described in [AQM+96] because in this paper we do not consider regular expressions or "wildcards" appearing within the path expression. This restriction is further discussed in Section 3.5.

3.2 Query Language

Lorel is the query language supported by the Lore DBMS. Lorel is an extension of OQL supporting declarative path expressions for traversing semistructured data, and extensive automatic coercion for handling heterogeneous and/or typeless data without generating errors. Although Lorel offers much syntactic sugar over OQL that is convenient in practice (including the shorthands mentioned earlier), in this paper we write our queries without any shorthands in order to be very explicit and enable understanding for those familiar with OQL but unfamiliar with Lorel. As a simple illustrating example, consider the following query, which asks for all of the young members of the Database Group.[2] In the remainder of the paper we refer to this query as "Query 1". The result of the query over the database of Figure 1 is shown, with indentation to represent the subobject relationship.

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

This very simple query is used as an example throughout the paper to illustrate our basic query optimization techniques. Our optimizer does handle more complex queries, such as the following one which asks for the set of names of all pairs of group members who work on the same project and whose age differs by more than ten years.

```
Select m1.Name, m2.Name
From   DBGroup.Member m1, m1.Project p1,
       DBGroup.Member m2, m2.Project p2
Where  p1 = p2 and exists a1 in m1.Age: exists a2 in m2.Age: abs(a1 - a2) > 10
```

3.3 Lore Query Processing

The general architecture of the Lore system looks very much like a traditional DBMS [MAG+97]. The components most relevant to this paper are the query compilation and query execution components shown in Figure 2. After a query is *parsed*, it is *preprocessed* to convert the Lorel shorthands into a more traditional OQL form. The *logical query plan generator* then creates a single logical query plan. The logical query plan is a tree composed of *logical operators* describing a high-level

[2]The existential quantification in the `Where` clause is necessary since in a semistructured database a `Member` object could conceivably have many `Age` subobjects. A shorthand in Lorel allows simply "`x.Age < 30`" as the `Where` clause, which is preprocessed automatically into the query as shown [AQM+96].

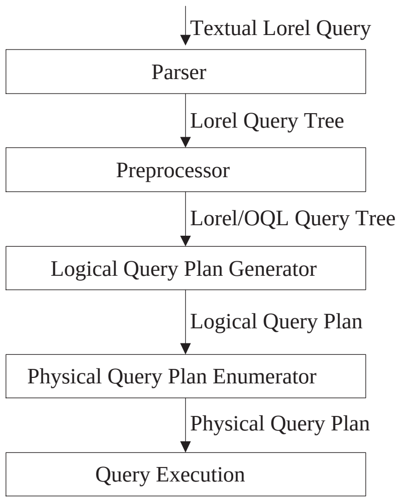

Figure 2: The Lore query processor

execution strategy for the query. As we will show in Section 5, generating logical query plans is fairly straightforward, but special care needed to be taken to ensure that the logical query plans are flexible enough to be transformed into vastly different physical query plans. The "meat" of the query optimizer occurs in the *physical query plan enumerator*. This component uses statistics and some metadata in order to transform the logical query plan into the estimated best physical plan that lies within our search space. A 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]. With iterators, execution begins at the top of the tree-structured query plan, with each node in the plan requesting a "tuple" at a time from its children and performing some operation on the tuple(s). After a node completes its operation, it passes a resulting tuple up to its parent. For many physical query plan operators, an iterator approach avoids creation of temporary data. 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 traditional 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 the path to an object is as important as the value of the object. 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, allows us to locate all parents for 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 that are connected via a specified labeled edge. Our full text index, or Tindex, supports finding atomic string values that match an information-retrieval style pattern. In addition to these indexes, Lore's *DataGuide* [GW97] provides the functionality of a path index, or Pindex. Since a frequent use of Lore is as a read-only query engine for data imported from the World-Wide Web, we have found that the overhead of building, storing, and maintaining all of these indexes usually is worthwhile. However, as in any DBMS, the indexes built on a given database can be selected by the system administrator, and the optimizer only creates plans based on indexes that exist.

Value indexing in Lore requires some novel features due to Lore's non-strict typing system. When comparing values of different types, Lore always attempts to coerce the values into comparable types. The current indexing system deals with coercions involving integers, reals, and strings only, although it can easily be generalized. Further details on how the Vindex handles coercion can be found in [MAG+97].

3.5 Relevant Lorel Subsets

Like SQL and OQL, Lorel is a big language. Most of Lorel is functional within the Lore system, including some subqueries, aggregation and arithmetic operations, constructed results, a declarative update language, and view facilities [AGM+97, AMR+97]. Missing features at the time of writing include full support for arbitrary subqueries, `Groupby` and `OrderBy` clauses, external functions and predicates, and full regular expressions appearing within path expressions (although a useful subset of regular expressions is supported). Our query processor can generate correct "naive" physical query plans for the entire supported language. Our cost-based optimizer also operates on the full language except that it does not handle queries that contain regular expressions appearing within path expressions, nor does it consider the related issue of how to handle cycles within the data (but again, we emphasize that such queries are supported by our system through straightforward strategies). There has been good work on these topics [CCM96, FS98] and it is our hope to incorporate some of this work into our optimizer. In addition, the Tindex is new to Lore so it is not yet considered by the optimizer. As will be seen, the query optimization problem for semistructured data is sufficiently complex that it made sense to begin with these restrictions. Due to space constraints, we consider a further subset of the language in this paper for clarity of presentation, although the general optimization principles are evident.

4 Motivation

As in any declarative query language, there are many ways to execute a single Lorel query. In this section we motivate how queries over semistructured data can be optimized. We will continue to use Query 1 introduced in Section 3.2, and will 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 semistructured 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) and evaluate the `From` clause by processing each simple path expression in a forward manner, moving from one variable to the next. Once all variables appearing in the

From have been "bound", the Where clause is handled in a similar fashion. (Recall that an iterator approach is being used, so this process is repeated for all variable bindings.) This query execution strategy results in a depth-first traversal of the graph following edges that appear in the path expressions.

Another way to execute Query 1 is to first identify all objects that satisfy the Where clause 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 in reverse the path expressions appearing in the Where and then in the From. Since our implementation does not support parent (inverse) pointers, we use the Lindex to move from objects to their parents. 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. The advantage of this approach is that we start with objects guaranteed to satisfy the Where clause, and do not needlessly explore paths through the data only to find that the final atomic object does not satisfy a predicate in the Where. Bottom-up is not always better than top-down, however, since there could be very few paths satisfying the simple path expressions in the From but many objects satisfying the condition in the Where.

A third strategy for executing Query 1 is to evaluate some, but not necessarily all, of the From clause in a top-down fashion and create a set of "valid" objects. Then directly identify those atomic objects that satisfy the Where using the Vindex and traverse up, via the Lindex, to the same point as the top-down exploration. By intersecting the sets of objects and combining traversed paths we find the result of the query. We call this a *hybrid* plan, since it operates both top-down and bottom-up, meeting in the middle of a path expression. This type of query plan can be optimal when the fan-in degree of the reverse evaluation of a path expression becomes too large at about the same time that the fan-out degree in the forward evaluation of the path expression becomes too 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 one path expression. The actual search space of plans for this simple query is much larger, as we will illustrate in Section 8, and more complicated queries with multiple path expressions naturally have an even larger variety of candidate plans.

To make things more concrete, suppose we are processing the query "`Select x From A.B x Where exists y in x.C: y = 5`", which is similar to Query 1. Example databases where each type of query plan described above would be a good strategy for this query appear in Figure 3. 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. The second database has many `A.B.C` paths, but only a single path satisfying the Where, so bottom-up is a good candidate. Finally, in the third database top-down execution would visit all the leaf nodes, but only a single one satisfies the query. Bottom-up would identify the single object satisfying the Where, but would visit all of the nodes in the upper right portion of the database. In this case, a hybrid plan where we use top-down execution to get the set of all `A.B` objects, then bottom-up execution for one level, then finally intersect the sets, would be a good query execution strategy.

Consider how very different just these three example plans are. In top-down we have a forward evaluation of all path expressions in the From and then evaluation of the Where. In bottom-up, we first handle the Where and then a reverse evaluation of all path expressions. Finally, the hybrid approach can evaluate either the From or the Where first, but sets of objects must be created and

Query: Select x From A.B x where exists y in x.C: y = 5

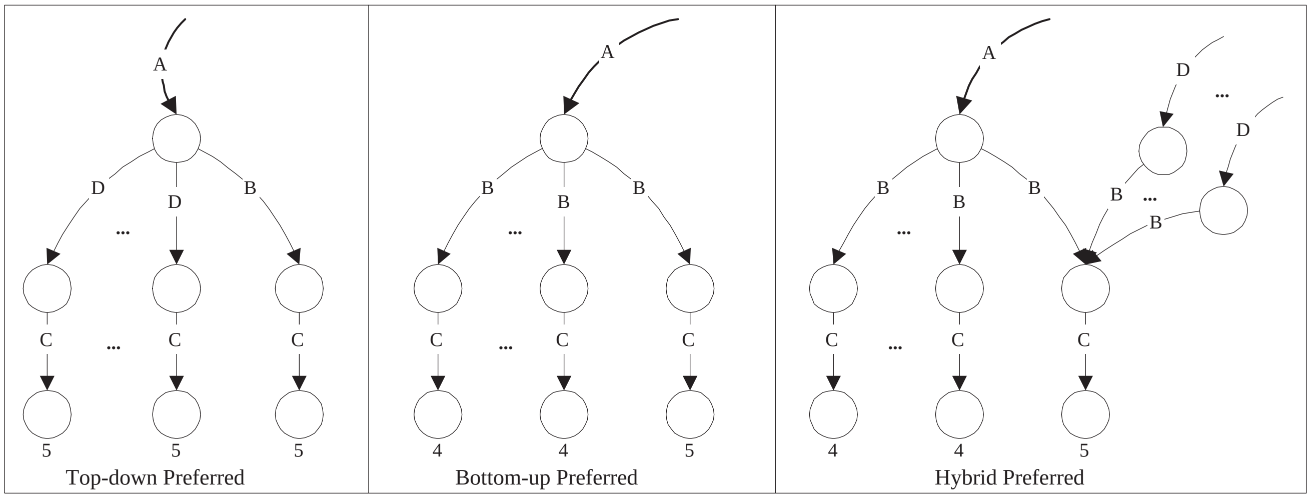

Figure 3: Different databases and some good query strategies

intersected. Each of these approaches results in a query plan that has a substantially different shape from the others, and each is the optimal plan for a particular database. One important contribution of this paper is in the design of our logical query plans, which allow simple transformations into any number of vastly different physical query plans.

5 Logical Query Plans

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, variable y in the simple path expression $x.l\ y$ is said to be *bound* after an l-labeled subobject o of x has been assigned to y. We also say that *o is bound to y.*

Definition 5.2 (**Evaluation**) An *evaluation* of a query (sub)plan is a list of all variables appearing in the (sub)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 query results based on these evaluations. Figure 4 presents some of the logical query plan operators used in Lore. Due to space constraints we omit several operators such as those used for arithmetic, aggregation, update, and view operations.

Recall that 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 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

Node	Parameters	Description
Project	Variable	Project out *Variable*.
Select	Predicate	Apply the *Predicate* to the current evaluation.
CreateSet	PrimVar, SecVar, DestVar	Accumulate a set of intermediate structures placed in the destination variable, *DestVar*, where each intermediate structure tracks a primary variable, *PrimVar*, and a set of secondary variables, *SecVar*.
Set	LeftPlan, RightPlan, SetOp	Perform a set operation *SetOp* (such as *union*, *intersect*, or *except*) using two sets of CreateSet structures passed up from the children nodes.
Glue	LeftPlan, RightPlan	Used to connect two independent subplans, where either subplan could be executed first.
Name	Name, DestVar	Find the named object, *Name*, and place the object into the destination variable, *DestVar*.
Chain	LeftPlan, RightPlan	Connector used between two components of a path expression.
Discover	SimplePathExpression	Used to discover information about the current database based on the *SimplePathExpression*.
Exists	Variable	Ensure that there is a binding for *Variable*.

Figure 4: Logical query plan operators

bottom-up plan we first use the `Vindex` to find all objects that satisfy the predicate "`y < 30`", and then use the `Lindex` to find all parents pointing to this object with label `Age` that is, we perform (b) before (a). In the top-down strategy we do the opposite, first finding an `Age` child of x and then testing the condition.

In fact, all queries can be broken into independent components where the execution order of the components is not fixed in advance. When creating a logical query plan, the *Glue* operator is used to connect two such components of the query. At a high level, each query is put into the form shown in Figure 5 (we abbreviate Path Expressions as PE in the figure). Among other things, this coarse outline indicates that execution of the `From` clause is completely separate from the `Where`, and that either subplan could potentially be executed first. In special cases we know that a certain order is almost always preferable, so as a heuristic to prune the search space we provide a flag in the *Glue* operator indicating whether or not to consider swapping the execution order of the left and right subplans when enumerating physical plans.

In the logical query plan, 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`" which has the logical query subplan shown in Figure 6. 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 asking both children for their most efficient way of executing their subplans and then joining them together in

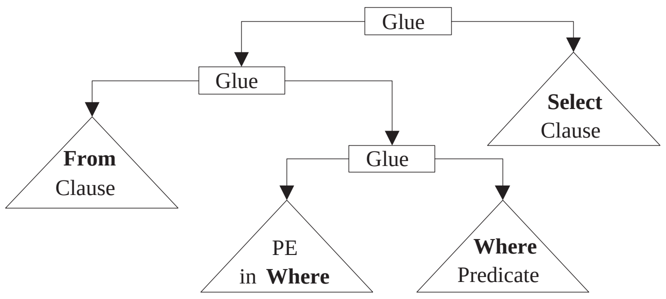

Figure 5: The general form of a logical query plan

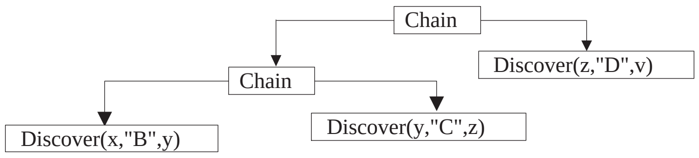

Figure 6: Representation of a path expression in the logical query plan

some fashion, or by a method called *TargetSet* that we discuss later.

As mentioned in Section 3.1, a *name* is a reference to a single object and is used as an entry point into the database. We use a special logical query plan operator *Name* to discover information about named objects. Names have several special properties including the fact that they are unique, and they are stored efficiently due to their frequent access. Also, since the database is divided into *workspaces* within a database (to handle views and private areas within which individual users can work) the *Name* operator is a convenient place to reference the workspace over which a path expression is being evaluated.

The *Select* and *Compound* operators are used for the `Where` clause to represent single comparisons and compound predicates (*and*, *or*) respectively. The *Exists* operator is used to handle existentially quantified variables that appear in the `Where` clause. The *Exists* operator is always placed directly above the *Discover* node for the relevant variable. The *Project* operator projects out a single variable and typically appears as the topmost node in a logical query plan.

The *CreateSet* operator creates a set of intermediate structures based upon the evaluations of its children. It is needed for the set operations (*union*, *intersect*, *except*), which operate over sets of evaluations, and is similar to creating a temporary table in a relational query plan. One variable, bound by the subplan rooted at the child of the *CreateSet* operator, is designated as *Primary.* A list of variables, which also must be bound below, is designated as *Secondary.* Each intermediate structure associates, for one object bound to the primary variable, a set of bindings for each secondary variable. In Section 6 we will explain why the *CreateSet* operator creates an intermediate structure with secondary variable bindings and not just a straightforward set of objects.

Figure 7 shows the complete logical query plan for Query 1. 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 guarantees that y is existentially quantified. A *Glue* node separates the discovery of the existential in the `Where` from the actual predicate test,

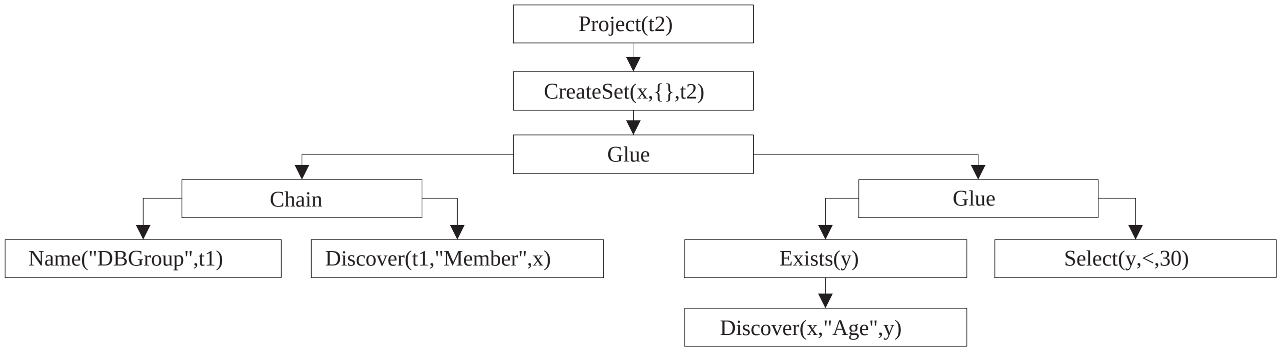

Figure 7: A complete logical query plan

allowing either operation to occur first in the physical query plan. Because the semantics of Lorel requires a set of objects to be returned, the *CreateSet* node gathers the satisfying evaluations at the top of the query plan and projects the result from that set.

6 Physical Query Plans

Now we describe the physical query plan operators and give some examples of complete physical query plans. Figure 8 shows some of the physical query plan operators used in Lore. Again, due to space constraints we omit several operators such as those used for arithmetic, update, and view operations. In Section 8, we will specify the method by which logical query plans are transformed into physical query plans. Recall that our physical query plan nodes act as iterators, where each node in the plan requests a "tuple" at a time from its children and performs some operation on the tuple(s). Here the "tuples" that our query plans operate over are evaluations (Definition 5.2), which are sets of bindings for variables in our query.

Many of the physical operators in Figure 8 are exactly the same as their relational counterparts. For example, *Select* simply applies a predicate to the current evaluation, and *Project* projects out a single variable in the current evaluation. The *Join* operator is similar to the nested-loop join in relational systems: for every successful evaluation of the *LeftPlan*, the *RightPlan* is called to exhaustion. Each pair of evaluations between the left and right plans is passed up to the *Join's* parent. The *Compound* operator supports *and* and *or* in the usual (short-circuited) manner.

In physical query plans, there are six ways to identify information stored in the database:

1. *Scan*: The *Scan* operator is similar in functionality to a relational scan. However, instead of scanning the set of tuples in a relation, our scan returns into y all objects that are subobjects of the complex object x via an edge labeled l.

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

Name	Parameters	Function
Project	Variable	Project out *Variable*.
Select	Predicate	Apply the *Predicate* to the current evaluation.
Join	LeftPlan, RightPlan	Straightforward implementation of the relational nested-loop join.
Compound	LeftPlan, RightPlan, CoOp	Links two components of a compound predicate, where the operator, *CoOp*, can be either *and* or *or*.
Scan	x, l, y	Place all subobjects of x that have label l into y.
Lindex	x, l, y	Place all l-labeled parents of the object in y into x.
TargetSet	PathExpression, DestVar	Using the DataGuide's target set, put all objects reachable via *PathExpression* into *DestVar*.
FindObj	x, l, y	Find all edges in the database with label l and place the parent into x and the child into y.
Vindex	Label, Op, Value, DestVar	All atomic values satisfying *Op Value* and that have an incoming edge labeled *Label* are placed into *DestVar*.
NamedObj	SourceVar, Name	Confirm that the object in *SourceVar* is the Named object *Name*.
Once	Variable	Ensure that an object is only assigned to *Variable* a single time.
CreateSet	PrimVar, SecVar, DestVar	Accumulate a set of intermediate structures, where each intermediate structure tracks information about a primary variable, *PrimVar*, and a set of secondary variables, *SecVar*.
Set	LeftPlan, RightPlan, SetOp	Perform a set operation *SetOp* (such as *union*, *intersect*, or *except*) using two sets of CreateSet structures passed up from the children nodes.
Deconstruct	SourceVar	Take the intermediate structure in *SourceVar* and decompose it into its components.
ForEach	SourceVar	Take the set of objects in *SourceVar* and iterate over them one at a time.
Aggregation	SourceVar, Op, DestVar	Perform the aggregation operation, *Op*, over *SourceVar*. In addition, this operator can be used to ensure that at least one object was successfully bound to *SourceVar*.

Figure 8: The physical query plan operators

3. *TargetSet*: Lore maintains a dynamic "structural summary" of the current data 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 set of oids is called the *Target Set* for a path expression. The *TargetSet* operator places all objects reachable via *PathExpression* into a destination variable *DestVar*.

4. *FindObj*: The *FindObj* operator finds all parent-child pairs connected via an edge labeled l. *FindObj* allows us to efficiently locate edges whose label appears infrequently in the database. The *FindObj* operator is implemented using Lore's edge index (Section 3.4).

5. *NamedObj*: The *NamedObj* operator simply verifies that the object in variable *SourceVar* is the named object *Name*.

6. *Vindex*: The *Vindex* operator accepts a *Label*, an operator *Op*, and a *Value*, and finds all atomic objects that satisfy the "*Op Value*" condition and have an incoming edge labeled *Label*. Each satisfying object is placed in the *DestVar*.

As an example that uses some of the operators discussed above, consider the path expression "`A.B x, x.C y`" and four possible subplans as shown in Figure 9. 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 3. If we assume that we already have a binding for y then we can use the second plan, the "Lindex & NamedObj 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 query execution strategy. In the "TargetSet Plan", we use the *TargetSet* operator which allows us to directly obtain the set of objects reached via the given path expression. In the "FindObj Plan", we directly locate all parent-child pairs connected via a `B` edge using the *FindObj* operator. We then need to confirm that the parent object is the named object `A`, and *Scan* for all of the `C` subobjects of the child object.

The *Set* operator is key to the hybrid query execution strategy introduced in Section 3. This operator allows one subplan to evaluate a portion of the query and obtain bindings for a set of variables, say V, and another subplan to obtain bindings for another set of variables, say W. If $V \cap W$ contains one variable, then both plans require that a single variable be bound to certain objects. Furthermore, if the plans are otherwise *independent*, meaning one does not provide a binding that the other uses, then by creating the evaluations for both subplans and intersecting the sets of objects for the shared variable, we efficiently obtain the complete evaluations that are valid between both sets. While it is possible to combine the subplans without a set operation, it is very inefficient to do so—it would be analogous to executing an uncorrelated subquery in SQL once for each tuple produced by the outer query.

In some situations, the temporary data created by *CreateSet* may need to contain more than just the bindings for the "primary" variable that we will perform the *Set* operation over. Without additional information, those variables that are not the primary variable will lose their bindings after successive calls to *CreateSet*'s subplan. To accommodate this situation, *CreateSet* actually has three arguments: (1) the primary variable over which the set operation is performed; (2) the set of secondary variables that will be tracked in conjunction with a primary variable; and (3) the variable to hold the resulting set. This structure allows us to recall, for a given object o bound to

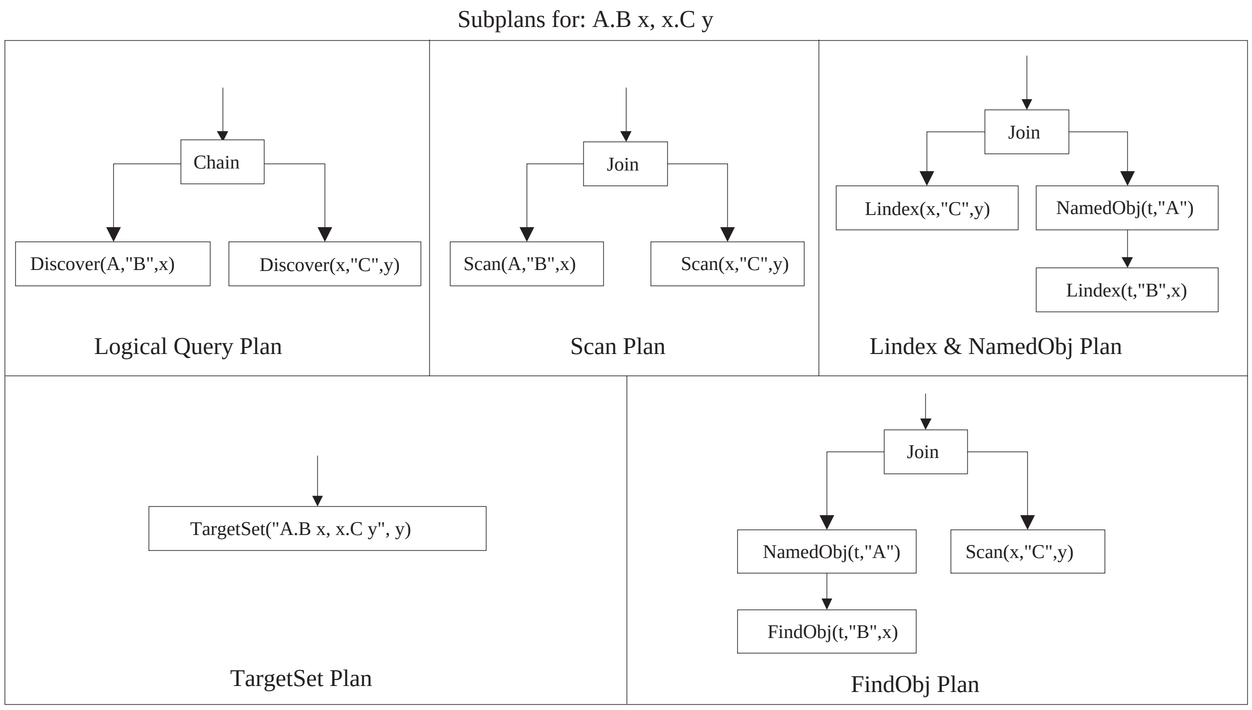

Figure 9: Different physical query plans

the primary variable, a set of objects that were bound to a secondary variable at the same time that o was bound to the primary. Needless to say, we have been careful to encode this temporary data compactly.

The *Set* operator performs any of the standard set operations, *union*, *intersect*, and *except*, over the two sets of intermediate structures passed up from its children. The object resulting from the set operation is another set of intermediate structures. The primary variable of the incoming intermediate structure is used to perform the actual set operation, while the resulting secondary structure is the union of the two secondary structures for the satisfying primary objects.

The *Aggregation* operator can be used both for standard aggregation operations (not discussed here) and also to ensure that a variable satisfies an existential quantification: aggregation with the *Exists* operation is used in top-down evaluation, and continues to call its subplan until a single binding has been found for its *SourceVar*. Subsequent calls to the aggregation operation with the same evaluation as the previous call will result in no evaluation being passed up, since the existential clause has already been satisfied. The *Once* operator also is used to ensure that a variable is existentially quantified, but *Once* is used during bottom-up evaluation and appears directly above the *Lindex* node that binds the existentially quantified variable. The *Once* operator only allows an evaluation to be passed to its parent node if the object bound in *Variable* has never been seen before.

The *Deconstruct* operator is the reverse operation of *CreateSet*. It accepts a *SourceVar* that holds an intermediate result from the *CreateSet* operation, and it decomposes the set into its components by placing, one at a time, the primary objects and corresponding sets of secondary objects into

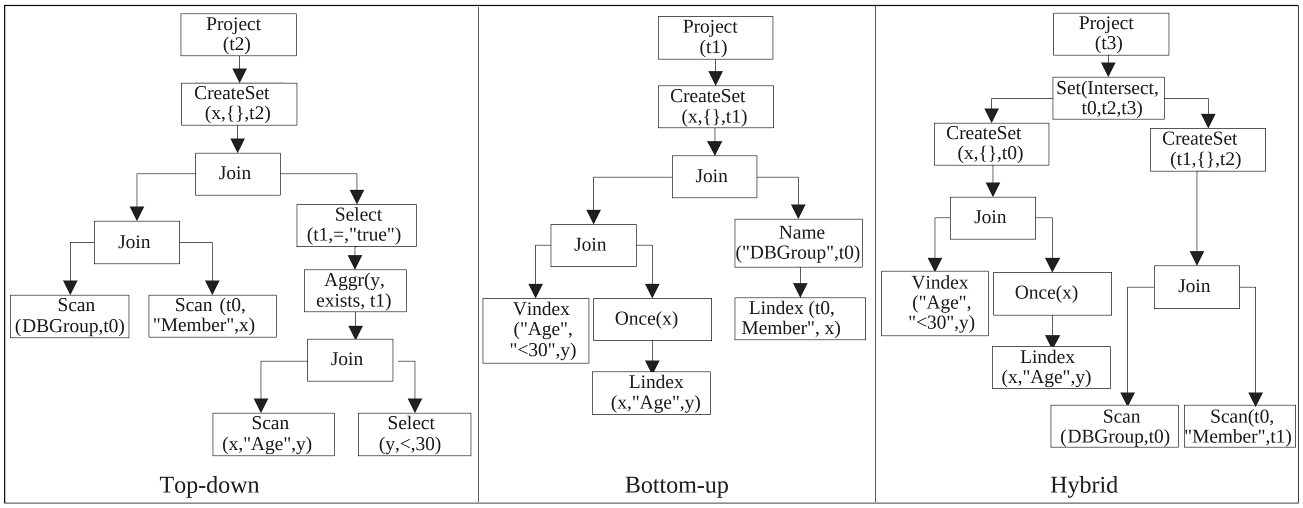

Figure 10: Three complete physical query plans

their original variables. After decomposition, a secondary variable will hold the set of secondary objects that are seen while visiting the primary object, so it is the responsibility of the *ForEach* operator to take a secondary set of objects and iterate through all of the members. Thus, we use the *CreateSet* operator to create sets of objects in order to apply intersection or union operations, then we use *Deconstruct* and *ForEach* to decompose the result and continue query processing on the original objects. In Figure 10 we give three complete physical query plans for Query 1. Each plan corresponds to one of the strategies that we introduced in Section 4. Again, we emphasize that these are only three representative plans, while numerous others are possible and are considered by our plan enumerator.

7 Statistics and Cost Model

As with any cost-based query optimizer, we need to establish a metric by which we will 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 predicated 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 and our relatively simple cost calculations, experiments presented in Section 9 validate that our cost model is reasonably accurate. Nevertheless, refining and expanding the cost model is an area where we intend to invest significant future effort.

7.1 Statistics

Our query optimizer must have access to statistical information about the size, shape, and range of values within an OEM database in order to estimate the cost of physical query plans. Although the lack of a schema makes it appear difficult to collect and store statistics, it turns out that the *DataGuide* [GW97], introduced earlier in Section 6, is the perfect vehicle. Roughly, the DataGuide, which is a “structural summary” of the database, maintains statistics for each distinct path expression by *annotating* corresponding single paths in the DataGuide. The statistics tracked for every path expression p 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 objects reachable via p, denoted $|p|$.
- The total number of distinct objects reachable via p, denoted $|p|_d$.

Further details on how the DataGuide computes and stores statistics for path expressions appear in [GW97].

As mentioned earlier, our cost metric is based on the estimated number of objects fetched during evaluation of the query. Thus, 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. For example, 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 an object reached by the path "`A.B x`" has. Alternatively, if we proceed bottom-up with a binding for y, then the optimizer must estimate the average number of parents via a `C` edge for an atomic object. 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 DataGuide statistics by $|p.l|/|p|_d$. Unfortunately, fan-in cannot be determined using the DataGuide statistics listed above. To enable calculating fan-in, we create a *reverse DataGuide* by logically fusing all atomic objects together into a single object and following parent pointers instead of child pointers, adding statistics as in the regular DataGuide. The reverse DataGuide allows us to estimate fan-in in the same manner as fan-out in the regular DataGuide, i.e., the fan-in for path p and label l is $|p.l|/|p|_d$ evaluated over the reverse DataGuide.

Our cost metric also uses statistics obtained from Lore's indexing components. From the Vindex we can estimate the number of atomic objects with an l-labeled incoming edge that satisfy a given predicate. From the Bindex we can determine the total number of times that an l-labeled edge appears within the database, which is necessary when estimating the cost of the *FindObj* operator. When estimating the number of atomic objects that will satisfy a given predicate, the usual formulas, e.g., those given in [SAC+79, PSC84], are insufficient in our semistructured environment due to the extensive type coercion that Lore performs. 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.

7.2 Cost Model

Each physical query plan (or subplan) 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 upon the costs assigned to the subplans rooted below the node, along with the cost for executing the node itself. For instance, the cost of the physical *Project* operator

Operator	IO Cost
Project	IOCost(Child)
Select	IOCost(Child)
Join	IOCost(Left) + |Left|$*$IOCost(Right)
Compound	IOCost(Left) + IOCost(Right)
Scan	$Fout_{PathOf(x),l}$
Lindex	2+ $Fin_{PathOf(y),l}$
TargetSet	$LengthOf(PathExpression)$+$\|PathExpression\|$
FindObj	$NumEdges(l)*2$
Vindex	$BLevel_{label,type1}$+ $Selectivity_1(label,Op,Value)$ +$BLevel_{label,type2}$+ $Selectivity_2(label,Op,Value)$
NamedObj	IOCost(Child) + |Child| $*$ 2
Once	IOCost(Child)
CreateSet	IOCost(Child) + |Child|$*$2$*$(1 + $\|SecVar\|$)
Set	IOCost(Left) + IOCost(Right) + *Struct*(Left) + *Struct*(Right)
Deconstruct	$\|SourceVar\| * (\sum_{x \in SecVar} \|x\| \ / \ \|SourceVar\|)$
ForEach	0
Aggregation	IOCost(Child) + 1

Figure 11: The cost formulas for physical query plan nodes

is the cost of its child along with the cost for projecting one variable from each evaluation returned by the child. In order to compute estimated cost recursively, at each node we must at the same time estimate the number of evaluations expected for that subplan. To decide if one (sub)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 very simplistic, but it does appear acceptable for the first version of our cost-based optimizer as shown by the performance results in Section 9, and we plan to refine and extend it as discussed earlier.

Our formulas to estimate I/O cost in number of page fetches are given in Figure 11. As discussed earlier, since we do not enforce clustering, we assume conservatively that each object fetch results in a page fetch. Due to space constraints, we do not include our formulas for estimated CPU cost and expected number of evaluations. As examples, let us consider the I/O formulas for the *Vindex* and *Lindex* operators. Because of type coercion, multiple B+-trees need to be accessed during a *Vindex* operation. Currently in Lore, three B+-trees are maintained to coerce the atomic types `string`, `integer`, and `real` among themselves. Depending upon the type of the value in the predicate, two of the B+-trees will be used [AQM$^+$96]. For each, we first traverse down to a depth at most $BLevel_{l,t}$, where $BLevel_{l,t}$ is the height for the Vindex B+-tree for label l and type t. We assume that each node in the tree requires a page fetch. We then add I/O cost based on the total number of estimated answers, encapsulated in the *Selectivity* function described below. The *Lindex* operator 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. The number of possible parents is based on the fan-in for the current path and label.

The following definitions are used in Figure 11. In the definitions, p is a path expression, x is a variable defined within a simple path expression, and l is a label.

- P: Page size.
- $Fout_{x,l}$: estimated l-labeled fan-out for any object bound to x. Formula is given in Section 7.1.
- $Fin_{x,l}$: estimated l-labeled fan-in for any object bound to x. Formula is given in Section 7.1.
- $|x|$: the number of objects expected to be bound to x. Determined by $|PathOf(x)|$ as obtained from the DataGuide, where *PathOf* takes a variable and returns the path expression that ends in that variable.
- $|plan|$: the expected number of evaluations returned by the subplan *plan*. Determined recursively; details omitted.
- *Selectivity*($Pred$): the estimated selectivity of predicate *Pred*. We use [SAC+79, PSC84] as a base with extensions to handle type coercion.
- *Struct*($plan$): the estimated number of object fetches to read and write the set of intermediate structures returned by subplan *plan*. Determined by $|plan.PrimVar| + (|plan.PrimVar| * (\sum_{x \in plan.SecVar} |x|))$.
- *NumEdges*(l): the number of edges with label l; supplied by the Bindex.
- *LengthOf*(p): the number of simple path expressions that comprise p.

8 Plan Enumeration

The search space of physical query plans for a single Lorel query is very large. For example, a path expression of length n can be viewed as an n-way join where we have several "join methods", and there may be many path expressions in a single query. In order to reduce the search time 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, based on the physical plans selected for its children. While this approach greatly reduces the search space (and seems to perform well; see Section 9), it still explores an exponentially large number of physical query plans. Thus, our optimizer currently uses the following heuristics to further prune the search space.

- *TargetSet* is used only when a path expression begins with a name, and no variable except the last is used elsewhere within the plan. This latter restriction is based on the fact that *TargetSet* only binds the last variable in the path expression so other variables would need to be discovered by some additional method.
- If a variable has been converted into a set by a *CreateSet* operator and we subsequently need to access its members, we always execute a *Deconstruct* operator and never consider plans that rediscover the original variable bindings.

- Applicable set operations are always preferred over rediscovering the same information; that is, we do not repeatedly execute "uncorrelated subqueries" as discussed in Section 6.
- The `Select` clause always executes last, since in nearly all cases it depends upon one or more variables bound in the `From` clause.
- The physical query plan will always execute either the complete `From` or complete `Where` clause before moving on to the other one.
- Multiple path expressions are handled in the order they are specified within the query. If the path expressions are independent, the optimizer will not attempt to re-order them.

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. For example, if one of the simple path expressions that appears in the `From` clause is not required by the `Where` clause, then it could be moved to after execution of the `Where`. This kind of post-optimization could "repair" any mistakes made by the last two heuristics mentioned above, although we have not yet worked out the details.

To specify how physical plan generation works, we need to consider how each logical plan node makes decisions, based upon decisions made by its earlier siblings and children. The state information maintained during the process includes the following for every variable in the query: (1) whether the variable is bound or not; (2) which plan operator has bound the variable; (3) the set of other plan operators that require use of the variable; (4) whether the variable is a primary or secondary variable in a set. When creating the physical query plan, each logical plan node receives the current state of the variables in the query and generates the optimal physical plan for that state. The new state of the variables and the optimal subplan are then passed to the parent.

Here we give an overview of how each type of logical plan node generates the optimal physical subplan.

- *Project*, *CreateSet*, *Set*, and *Compound*: simply returns the corresponding physical operator over the optimal physical query plan(s) for the child(ren).
- *Select*: If the variables appearing in the selection condition are all bound then we return the *Select* physical operator over the optimal plan for the child. If the variables are not bound, then we return the *Vindex* physical operator with no subplan.
- *Glue*: Creates the optimal physical query plan corresponding to the left-then-right child execution order and compares it with the optimal physical plan for the right-then-left child execution order. The cheaper plan is returned.
- *Name*: Returns a *Scan* physical operator if the *variable* has not been bound, otherwise returns a *NamedObj* physical operator.
- *Chain*: Creates the optimal physical query plan corresponding to the left-then-right child execution order and a second optimal physical query plan corresponding to the right-then-left child execution order. Compares the costs of these two against the cost of the *TargetSet* physical operator for the entire path expression rooted at the *Chain* node, and returns the cheapest of the three.

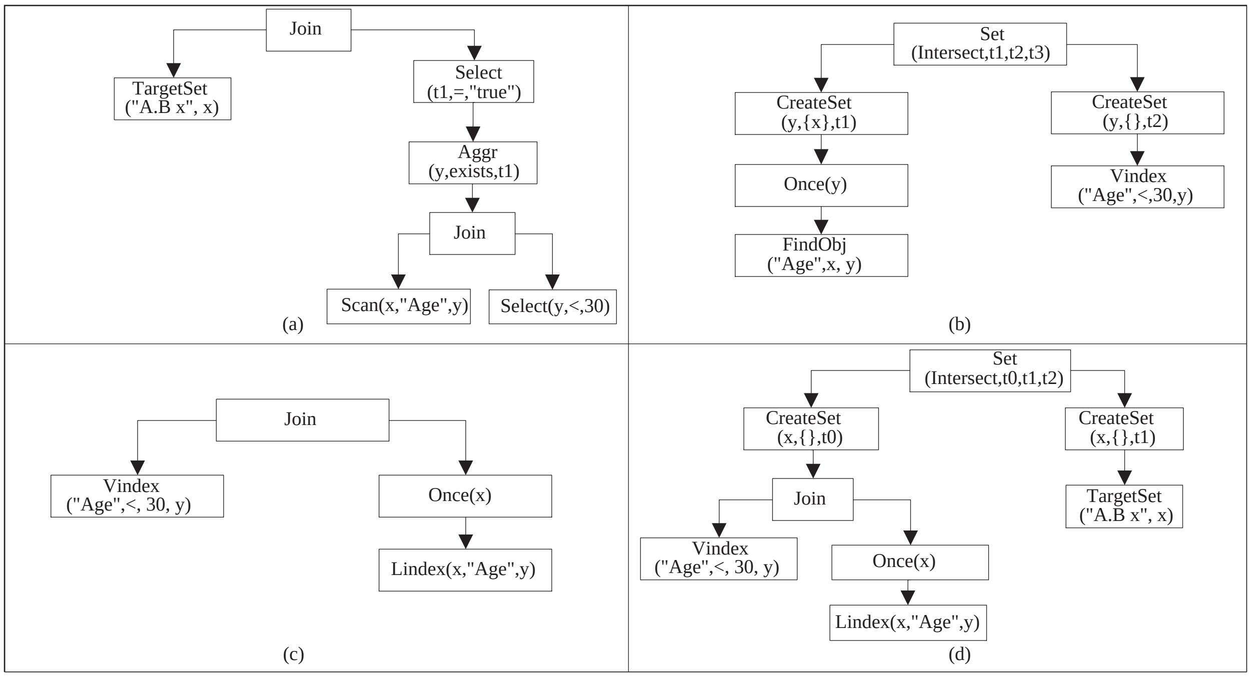

Figure 12: Sequence of possible transformations for Query 1 into a physical query plan

- *Discover*: If the *SourceVar* is bound then a *Scan* physical operator is created, otherwise a *Lindex* operator is created. This operator is then compared against the *FindObj* physical operator for the *Discover* node. The cheaper of the two is returned.
- *Exists*: Returns an *Aggregation* subplan[3] over the optimal physical plan for the child if the *Variable* is bound via a *Scan* physical operator, otherwise returns an *Exists* physical operator.

Two additional points about the process are worth noting. First, some logical operators such as *Glue* and *Chain* combine their children's optimal plans into a single plan. This can be done via either the *Join* or *Set* physical operators as indicated in Section 6. Second, a logical node may be unable to create any physical plan for a given state of the variables because it requires some variables to be bound before it may be executed. In this case, "no plan" is returned by the node, and a different choice can always be taken at a higher level in the plan.

As an example, 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 7. Note that this transformation is based on statistics describing the shape of the database and its atomic values. The topmost *Glue* node in Figure 7 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 12(a). After exploring left-then-right execution order, the topmost *Glue* node considers the right-then-left order. We examine the possible subplans for this ordering in more detail. The right child is another *Glue*

[3]When aggregation is used to existentially quantify a variable, a *Select* operator is placed directly above the *Aggregation* node to ensure that the existential condition is satisfied.

node which recursively follows the same procedure. Suppose that for this second *Glue* node, the left-then-right execution order resulted in the physical subplan shown in Figure 12(b), while the right-then-left execution order resulted in Figure 12(c). Suppose plan (c) is chosen as the cheaper one, since it does not require the expensive *CreateSet* operation. 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 12(d). Notice that in the right subplan for the topmost *Glue* node, the *Chain* node decided that the *TargetSet* operator is the best way to get all "`A.B x`" objects within the database, despite the fact that we already have a binding for x. This can happen when the estimated fan-in for x with label `B` is very high. As a final step the topmost *Glue* node must decide which query plan is optimal, either (a) or (d), and pass that plan up to its parent.

9 Performance Results

The optimization techniques described in this paper are fully implemented in Lore, including the physical operators, statistics, cost formulas, logical query plan generation, and physical query plan enumeration and selection. We now present some preliminary performance results showing that our cost model is reasonable and that the optimizer is choosing a "good" plan. We focus on comparing plans that use top-down, bottom-up, hybrid, and target set strategies, since in practice only very unusual database shapes result in optimal plans that use *FindObj*. 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 200K 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. Query running times are shown below.

We used two synthetically-generated databases to test our optimizer. The first database is a fairly well-structured tree of height 8 with a fan-out of 3 at each level. There are 2,188 total objects in the database. Labels are the same within each level but distinct between levels. The leaf nodes are populated with atomic integer values uniformly distributed between 1 and 10,000. The second database is a semistructured graph with a fan-out of up to 7 at each level and a straight-line depth (from the root to a leaf without traversing any sibling or back edges) of 12. Every 5th edge is a sibling or back edge, and there are 10,857 total objects in the database. The atomic values are 50% strings, chosen randomly from 1,000 words, and 50% integers between 1 and 10,000. For the remainder of this section we refer to the first database as *Tree* and the second as *Graph*.

9.1 Validating the Cost Model

To validate the relative accuracy of our cost model, we compare the estimated I/O cost of several physical query plans against their actual execution time. (Since CPU cost is used only rarely as a tie-breaker, we do not investigate its accuracy here.) We report on two different queries run over

	Point Query		Range Query		
Experiment #	1	2	3	4	5
Query Type	Top-down TargetSet Ok	Top-Down No TargetSet	Top-Down TargetSet Ok	Top-Down No TargetSet	Hybrid No TargetSet
Estimated Cost	7236	8803	8682	8803	10770
Actual Time	4.03138	4.23638	4.18361	4.30536	6.97613
Ratio (Est./Actual)	1794	2077	2075	2045	1544

Figure 13: Estimated cost vs. actual time on database *Tree*

	Point Query		Range Query		
Experiment #	6	7	8	9	10
Query Type	Top-down No TargetSet	Top-down TargetSet Ok	Hybrid TargetSet Ok	Hybrid No TargetSet	Top-down No TargetSet
Estimated Cost	133	88	3410	3763	80
Actual Time	0.1350	.12077	2.59154	2.73998	0.05187
Ratio (Est./Actual)	986	728	1317	1373	1542

Figure 14: Estimated cost vs. actual time on database *Graph*

each database, and we instructed the optimizer to consider a variety of different strategies.[4]. Both queries involved traversing a path expression of at least length 5 in the `From` clause, with several existentially quantified variables appearing in the `Where`. The first query is a point query, designed to produce very few results, while the second is a range query with 10% selectivity.

While the *Graph* database is much larger than the *Tree*, the range and point queries within the graph are much more selective than the corresponding queries within the tree since some paths in the graph lead to siblings or ancestors and not to atomic values. Recall that the estimated cost for a query plan is the estimated number of page fetches. The ratio between the estimated cost and actual execution time is our attempt to normalize the difference between the two, and the goal is to keep this value constant. As can be seen in Figures 13 and 14, the stability of the ratio is passable (especially when comparing different plans for a single query and database) but surely not perfect since we currently use such a simple metric for costing our query plans. On the good side, a very accurate result can be found in experiments 8 and 9, where the range query is applied over the *Graph*. The cost for experiment 9 is estimated as slightly higher (353 higher) than 8 because 9 does not use the *TargetSet* operator, but instead uses a sequence of *Scans*. The actual time reflects this increase.

[4]We can "hint" to the optimizer to favor certain strategies, in order to test different query plans for a single query, but unfortunately at the current time we cannot force a particular complete query plan. This explains the absence of pure bottom-up plans reported in Figures 13 and 14. We plan in the future to instrument the optimizer so that we can dictate certain plans, in order to enable more extensive experiments.

	Tree/Q1	Tree/Q2	Tree/Q3	Graph/Q4	Graph/Q5	Graph/Q6
Plan 1	0.01771 *	6.97613	69.5026	0.2131	2.59154	5.43245
Plan 2	4.03138	4.18361 *	5.30797 *	0.1207 *	2.73998	4.21658
Plan 3	4.23628	4.30536	5.44216	0.1350	0.5187 *	0.285 *

Figure 15: Execution time for queries and different plans

9.2 Importance of Choosing a Good Plan

Even for the relatively small databases *Tree* and *Graph*, choosing a good query plan is very important. Figure 15 presents actual query execution times for several queries, with three representative query plans for each one. As we can see, there can be up to two orders of magnitude difference in running time between plans. In all cases, our query optimizer chooses the starred (*) plan.

As examples, consider columns Tree/Q3 and Graph/Q6. In Tree/Q3 the query's `Where` clause was not very selective. The bottom-up plan, shown as Plan 1, used the *Vindex* operator to evaluate the `Where`, whereas the top-down plan (Plan 2) only explored those paths that satisfied the `From` and then checked the `Where`. In Graph/Q6, the query has a compound `Where` clause containing a disjunction. In Plans 1 and 2, the compound clause was handled by creating sets of satisfying objects for each predicate and doing a `union`. (The plans differ in that Plan 2 uses a *TargetSet* operator for the `From` and performs a second set operation, while Plan 1 does not.) In Plan 3, we instead use a short-circuiting version of the *Compound* operator, which results in its faster running time.

Needless to say, these experiments are very preliminary, but they indicate the importance of selecting a good plan and show that our optimizer does so. We plan to modify our optimizer so it can generate all possible plans (turning off our heuristics and dropping the greedy approach) so that we can test how close our selected plans are to the actual optimal one.

Acknowledgments

Many thanks to all members of the Lore project, past and present. We are especially grateful to Roy Goldman for his DataGuide efforts and feedback on this work, to Dallan Quass for his initial work on query processing in Lore, and to Qingshan Luo, Michael Rys, and Takeshi Yokokawa for their implementation efforts.

References

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

[AGM+97] S. Abiteboul, R. Goldman, J. McHugh, V. Vassalos, and Y. Zhuge. Views for semistructured data. In *Proceedings of the Workshop on Management of Semistructured Data*, pages 83–90, Tucson, Arizona, May 1997.

[AMR$^+$97] S. Abiteboul, J. McHugh, M. Rys, V. Vassalos, and J. Wiener. Incremental maintenance for materialized views over semistructured data. Technical report, Stanford University Database Group, 1997.

[AQM$^+$96] 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, November 1996.

[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.

[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.

[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. Technical report, AT&T Laboratories, 1998. To appear in *Proceedings of the Fourteenth International Conference on Data Engineering*, 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.

[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.

[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.

[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.

[Suc97] D. Suciu, editor. *Proceedings of the Workshop on Management of Semistructured Data.* Tucson, Arizona, May 1997.