Lore: A Database Management System for Semistructured Data*

Jason McHugh, Serge Abiteboul, Roy Goldman, Dallan Quass, Jennifer Widom
Stanford University
{mchughj,abitebou,royg,quass,widom}@db.stanford.edu
http://www-db.stanford.edu/lore

Abstract

Lore (for Lightweight Object Repository) is a DBMS designed specifically for managing semistructured information. Implementing Lore has required rethinking all aspects of a DBMS, including storage management, indexing, query processing and optimization, and user interfaces. This paper provides an overview of these aspects of the Lore system, as well as other novel features such as dynamic structural summaries and seamless access to data from external sources.

1 Introduction

Traditional database systems force all data to adhere to an explicitly specified, rigid schema. For many new database applications there can be two significant drawbacks to this approach:

- The data may be irregular and thus not conform to a rigid schema. In relational systems, null values typically are used when data is irregular, a well-known headache. While complex types and inheritance in object-oriented databases clearly enable more flexibility, it can still be difficult to design an appropriate object-oriented schema to accommodate irregular data.
- It may be difficult to decide in advance on a single, correct schema. The structure of the data may evolve rapidly, data elements may change types, or data not conforming to the previous structure may be added. These characteristics result in frequent schema modifications, another well-known headache in traditional database systems.

Because of these limitations, many applications involving *semistructured data* [Abi97] are forgoing the use of a database management system, despite the fact that many strengths of a DBMS (ad-hoc queries, efficient access, concurrency control, crash recovery, security, etc.) would be very useful to those applications.

As a popular first example, consider data stored on the World-Wide Web. At a typical Web site, data is varied and irregular, and the overall structure of the site changes often. Today, very few Web sites store all of their available information in a database system. It is clear, however, that Web users could take advantage of database support, e.g., by having the ability to pose queries involving data relationships (which usually are known by the site's creators but not made explicit). As a second example, consider information integrated from multiple, heterogeneous data sources [Com91, LMR90, SL90]. Considerable effort is typically spent to ensure that the integrated data is well-structured and conforms to a single, uniform schema. Additional effort is required if one or more of the information sources changes, or when new sources are added. Clearly, a database system that easily accommodates irregular data and changes in structure would greatly facilitate the rapid integration of heterogeneous databases.

This paper describes the implementation of the *Lore* system at Stanford University, designed specifically for managing semistructured data. The data managed by Lore is not confined to a schema, and it may be irregular or incomplete. In general, Lore attempts to take advantage of structure where it exists, but also handles irregular data as gracefully as possible. Lore (for *Lightweight Object Repository*[1]) is fully functional and available to the public.

Lore's data model is a very simple, self-describing, nested object model called *OEM* (for *Object Exchange Model*), introduced originally in the *Tsimmis* project at Stanford [PGMW95]. One of our first challenges was to design a query language for Lore that allows users to easily retrieve and update data with no fixed, known structure. *Lorel*, for *Lore Language*, is an extension of OQL [Cat94, BDK92] that introduces extensive type coercion and powerful path expressions for effectively querying semistructured data. OEM and Lorel are reviewed briefly in this paper; for details see [AQM+96].

Building a database system that accommodates semistructured data has required us to rethink nearly every aspect of database management. While the overall architecture of the system is relatively traditional, this paper highlights a number of components that we feel are particularly interesting and unique.

First, query processing introduces a number of challenges. One obvious difficulty is the absence of a schema to guide the query processor. In addition, Lorel includes a powerful form of navigation based on path expressions, which requires the use of automata and graph traversal techniques inside the database engine. The indexing of semistructured data and its use in query optimization is an interesting issue, particularly in the context of the automatic type coercion provided by Lorel. As will be seen, despite these challenges we are able to execute queries using query plans based primarily on familiar database operators. To accommodate semistructured data at the physical level (as well as support for multimedia data such as video, postscript, gif, etc.) we impose no constraints on the size or structure of atomic or complex objects. Meanwhile, however, the layout of objects on disk is tailored to facilitate browsing and the processing of path expressions.

Perhaps the most novel aspects of Lore are the use of *DataGuides* in place of a standard schema, and Lore's *external data manager*. A DataGuide is a "structural summary" of the current database that is maintained dynamically and serves several functions normally served by a schema. For example, DataGuides are essential for users to explore the structure of the database and formulate queries. They also are important for the system, e.g., to store statistics and

*This work was supported by the Air Force Rome Laboratories and DARPA under Contracts F30602-95-C-0119 and F30602-96-1-031, and by equipment grants from IBM and Digital Equipment Corporations.

[1] Originally, "lightweight" referred both to the simple object model used by Lore and to the fact that Lore was a lightweight system supporting single-user, read-only access. As will be seen, Lore is evolving towards a more traditional "heavyweight" DBMS in its functionality.

guide query optimization. Finally, because one of the motivations for using a DBMS designed for semistructured data is to easily integrate data from heterogeneous information sources (including the World-Wide Web), Lore includes an external data manager. This component enables Lore to bring in data from external sources dynamically as needed during query execution, without the user being aware of the distinction between local and external data.

We have chosen to implement Lore from scratch, rather than building an extension to an existing DBMS to handle semistructured data. Building our own complete DBMS allows us full control over all components of the system, so that we can experiment easily with internal system aspects such as query optimization and object layout. In parallel, however, we are implementing our semistructured data model and query language on top of the O_2 object oriented system [BDK92], in order to compare the implementation effort and performance against Lore. This paper focuses on Lore, although the O_2 implementation is discussed briefly.

1.1 Related Work

A preliminary version of the language Lorel was introduced in [QRS+95]. Details of the syntax and semantics of the current version of Lorel can be found in [AQM+96]. A comparison of Lorel against more conventional languages such as OQL [Cat94], XSQL [KKS92], and SQL [MS93] appears in [QRS+95]. Although the Lore system has been demonstrated [QWG+96], this is the first paper to describe implementation aspects of Lore.

The closest current system to Lore is *UnQL* [BDS95, BDHS96], which also is designed for managing semistructured data and uses a data model similar to OEM. While the UnQL query language is more expressive than Lorel, we believe it is less user-friendly. Furthermore, UnQL work has focused primarily on aspects of the query language and its optimizations and, so far, less on system implementation. A much earlier system, *Model 204* [O'N87], was based on self-describing record structures. As will be seen, the data model used in Lore is more powerful in that it includes arbitrary object nesting, and Lore's query language is richer than the language of Model 204. Thus, query processing in Lore is significantly different than in Model 204, which concentrated on clever bit-mapped indexing structures. Furthermore, to the best of our knowledge, Model 204 did not include concepts analogous to our DataGuides or external data.

There have been a number of other proposals that invent or extend query languages roughly along the lines of Lorel, or that integrate traditional databases with semistructured text data. Most of this work operates on strongly-typed data, or in some cases is designed specifically for the World-Wide Web. Examples include [BK94, BCK+94, CACS94, CCM96, CM89, KS95, LSS96, MMM96, MW95, MW93, YA94]. For a more in-depth comparison of these languages and systems against Lore, see [AQM+96].

1.2 Outline of Paper

Section 2 reviews the data model and query language used by Lore. Section 3 introduces the overall architecture and the individual components of the Lore system. Query and update processing, optimization, and indexing are considered in Section 4. Section 5 covers Lore's external data manager and DataGuides. Section 6 describes the various interfaces to Lore for developers, users, and application programs. Finally, Section 7 covers system status, describes how to obtain the Lore system, and discusses current and future work.

2 Representing and Querying Semistructured Data

To set the stage for our discussion of the Lore system, we first introduce its data model and query language. For motivation and further details see [AQM+96].

2.1 The Object Exchange Model

The *Object Exchange Model* (OEM) [PGMW95] is designed for semistructured data. 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 the 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 objects and as entry points into the database. In Figure 1, *DBGroup* is a name that denotes object &1. Any object that cannot be accessed by a path from some name is considered to be deleted.

In an OEM database, there is no notion of fixed schema. All the 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.

For an OEM object X and a label l, the expression $X.l$ denotes the set of all l-labeled subobjects of X. If X is an atomic object, or if l is not an outgoing label from X, then $X.l$ is the empty set. Such "dot expressions" are used in the query language, described next.

2.2 The Lorel Query Language

In this subsection we introduce the *Lorel* query language, primarily through examples. Lorel is an extension of OQL and a full specification can be found in [AQM+96]. Here we highlight those features of the language that have an impact on the novel aspects of the system—features designed specifically for handling semistructured data. Many other useful features of Lorel (some inherited from OQL and others not) that are more standard will not be covered.

Our first example query introduces the basic building block of Lorel: the *simple path expression*, which is a name followed by a sequence of labels. For example, `DBGroup.Member.Office` is a simple path expression. Its semantics consists of the set of objects that can be reached starting with the `DBGroup` object, following an edge labeled `Member`, then following an edge labeled `Office`. Range variables can be assigned to path expressions, e.g., "`DBGroup.Member.Office X`" specifies that X ranges over the set of offices. Path expressions also can be used directly, in an SQL style, as in the example.

The example query retrieves the offices of the older members of the group. The query, along with its answer for our sample database in Figure 1, follow. Note that in the query result, indentation is used to represent graph structure.

```
QUERY
   select DBGroup.Member.Office
   where DBGroup.Member.Age > 30
```

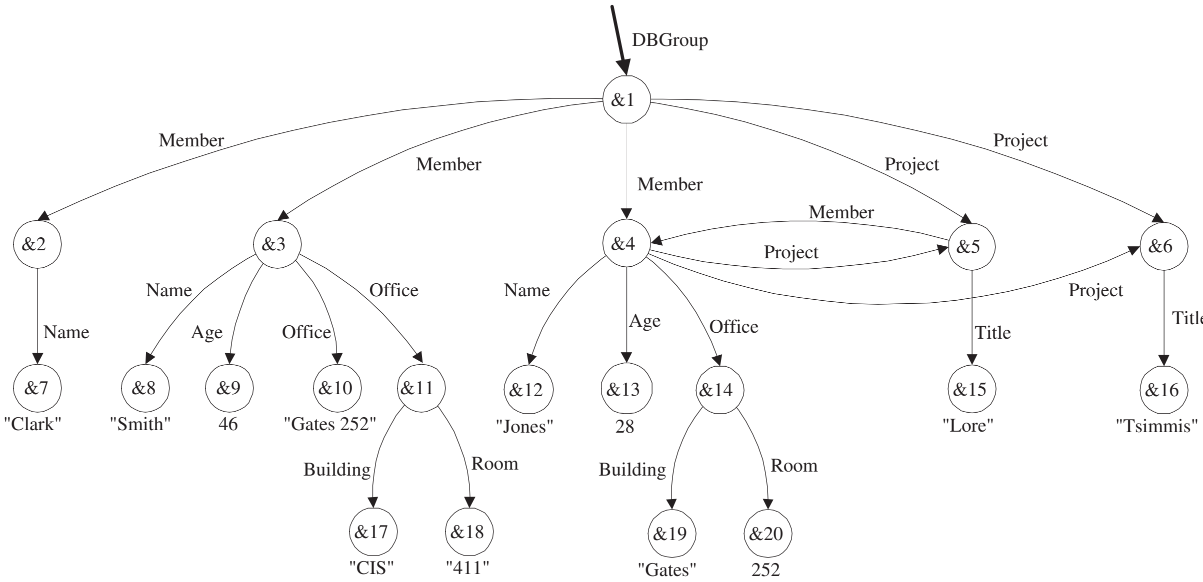

Figure 1: An OEM database

```
RESULT
   Office "Gates 252"
   Office
      Building "CIS"
      Room "411"
```

The database over which the query is evaluated presents a number of irregularities, as discussed earlier. A guiding principle in Lorel is that, to write a query, one should not have to worry about such irregularities or know the precise structure of objects (e.g., the structure of offices), nor should one have to bother with precise types (e.g., the type of *Age* is integer). This query will not yield a run-time error if an *Age* object has a string value or is complex, or if *Age*s or *Office*s are single-valued, set-valued, or even absent for some group members. Indeed, the above query will succeed no matter what the actual structure of the database is, and will return an appropriate answer.

The Lore query processor rewrites queries into a more elaborate OQL style. For example, the previous query is rewritten by Lore to:

```
select O
from DBGroup.Member M, M.Office O
where exists A in M.Age : A > 30
```

The Lore system then executes this OQL-style query, incorporating certain features such as special coercion rules (see Section 4.3) for the comparison $A > 30$.[2]

Note that a `from` clause has been introduced in the rewritten version of the query. (Omitting the `from` clause is a minor syntactic convenience in Lorel; a similar shorthand was allowed in *Postquel* [SK91].) Also note that the comparison on *Age* has been transformed into an existential condition. This transformation occurs because all properties are set-valued in OEM. Thus, the user can write `DBGroup.Member.Age > 30` regardless of whether *Age* is known to be single-valued, known to be set-valued, or unknown. We will see in Section 4 that an important first step of query processing in Lorel is rewriting the query into an OQL-style as above.

[2] We also are implementing Lorel on top of the O_2 system based on this translation to OQL; see Section 7 for a brief discussion.

Lorel offers a richer form of "declarative navigation" in OEM databases than simple path expressions, namely *general path expressions*. Intuitively, the user loosely specifies a desired pattern of labels in the database: one can specify patterns for paths (to match sequences of labels), patterns for labels (to match sequences of characters), and patterns for atomic values. A combination of these three forms of pattern matching is illustrated in the following example:

```
QUERY
 select DBGroup.Member.Name
 where DBGroup.Member.Office(.Room%|.Cubicle)?
          like "%252"

RESULT
   Name "Jones"
   Name "Smith"
```

Here the expression *Room%* is a label pattern that matches all labels starting with the string *Room*, e.g., *Room*, *Rooms*, or *Room68*. For path patterns, the symbol "|" indicates disjunction between two labels, and the symbol "?" indicates that the label pattern is optional. The complete syntax is based on regular expressions, along with syntactic wildcards such as "#", which matches any path of length 0 or more. Finally, "`like %252`" specifies that the data value should end with the string "252". The `like` operator is based loosely on SQL. We also support `grep` (similar to Unix) and `soundex` for phonetic matching.

During preprocessing, simple path expressions are eliminated by rewriting the query to use variables, as in our first example. It is not possible to do so with general path expressions, which require a run-time mechanism (described in Section 4.2). Indeed, note that if the database contains cycles, then a general path expression may match an infinite number of paths in the data. When trying to match a general path expression against the database, we match through a cycle at most once, which appears to be a reasonable simplification in practice.

We conclude with two more examples that illustrate advanced features of the language. The following query illustrates subqueries and constructed results. It retrieves the names of all members of the Lore project, together with titles of projects they work on other than Lore.

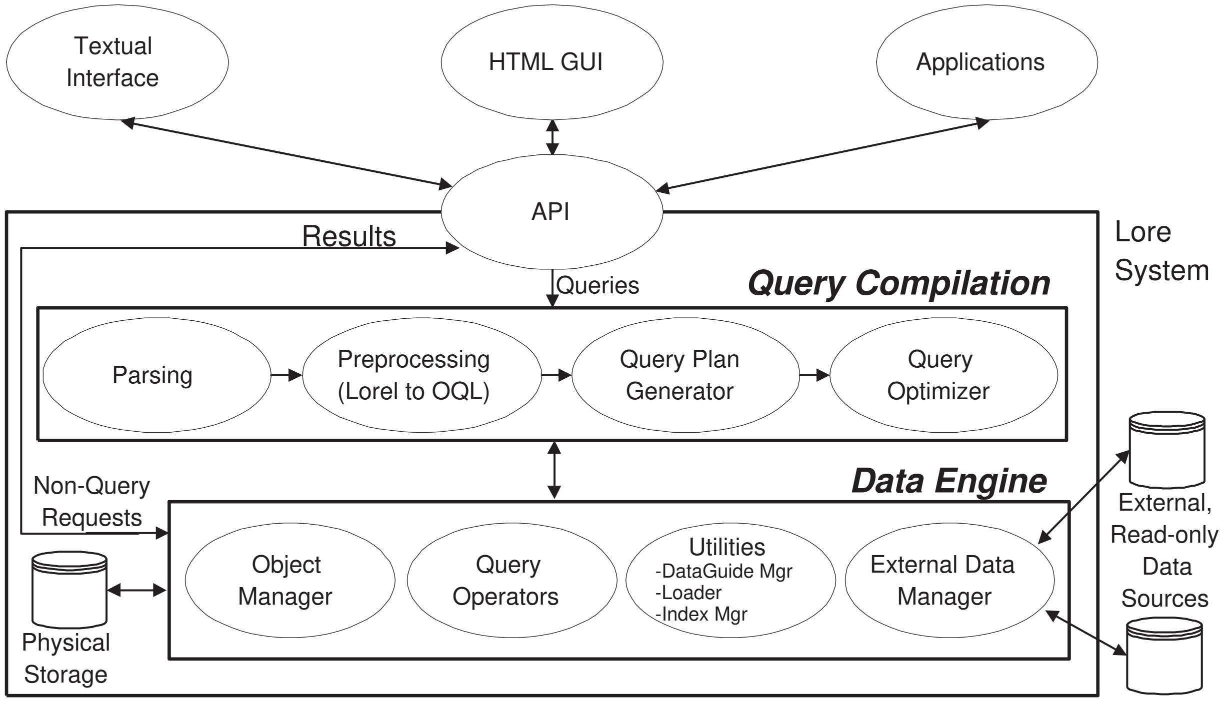

Figure 2: Lore architecture

```
QUERY
  select M.Name,
         ( select M.Project.Title
           where M.Project.Title != "Lore" )
  from DBGroup.Member M
  where M.Project.Title = "Lore"

RESULT
  Member
      Name "Jones"
      Title "Tsimmis"
```

Over a larger database, this query would construct one *Member* object for each group member in the result, containing the member's *Name* and a *Title* for each qualifying project.

A Lore database is modified using Lorel's declarative update language, as in the following example:

```
update P.Member +=
    ( select DBGroup.Member
      where DBGroup.Member.Name = "Clark" )
from DBGroup.Project P
where P.Title = "Lore" or
      P.Title = "Tsimmis"
```

This update adds all group members named Clark as members of the Lore and Tsimmis projects. Intuitively, the `from` and `where` clauses are first evaluated, providing bindings for P. For each binding, the expression "`P.Member +=`" specifies to add `Member` edges between P and every object returned by the subquery. In general, the update language supports the insertion and removal of edges, the creation of new vertices (objects), and the modification of atomic values and name assignments. (As mentioned earlier, object deletion is by unreachability, i.e., garbage collection, so there is no explicit delete operation.)

Lorel also offers grouping and aggregate functions in the style of OQL, external functions and predicates, and a powerful bulk loading facility that allows merging new data into an existing database. There is also a means of attaching variables to certain objects on paths, or even to the labels or paths themselves (in the style of the attribute and path variables of [CACS94]), which yields a rich mechanism for structure discovery. Such features, described in [AQM+96], are beyond the scope of this paper.

3 System Architecture

The basic architecture of the Lore system is depicted in Figure 2. This section gives a brief introduction to the components that make up Lore. More detailed discussions of individual components appear in subsequent sections.

Access to the Lore system is through a variety of applications or directly via the Lore Application Program Interface (API). There is a simple textual interface, primarily used by the system developers, but suitable for learning system functionality and exploring small databases. The graphical interface, the primary interface for end users, provides powerful tools for browsing query results, a DataGuide feature for seeing the structure of the data and formulating simple queries "by example," a way of saving frequently asked queries, and mechanisms for viewing the multimedia atomic types such as `video`, `audio`, and `java`. These two interface modules, along with other applications, communicate with Lore through the API. Details of interfaces are discussed in Section 6.

The Query Compilation layer of the Lore system consists of the parser, preprocessor, query plan generator, and query optimizer. The parser accepts a textual representation of a query, transforms it into a parse tree, and then passes the parse tree to the preprocessor. The preprocessor handles the transformation of the Lorel query into an OQL-like query (recall Section 2.2). A query plan is generated from the transformed query and then passed to the query optimizer. In addition to doing some (currently simple) transformations on the query plan, the optimizer also decides whether the use of indexes is feasible. The optimized query plan is then sent to the Data Engine layer.

The Data Engine layer houses the OEM object manager, query operators, external data manager, and various utilities. The query operators execute the generated query plans and are explained in Section 4. The object manager functions as the translation layer between OEM and the low-level file constructs. It supports basic primitives such as fetching an object, comparing two objects, performing simple coercion, and iterating over the subobjects of a complex object. In addition, some performance features, such as a cache of frequently accessed objects, are implemented in this component. The index manager, external data manager, and DataGuide manager are discussed in Sections 4.3, 5.1, and 5.2 respectively. Finally, bulk loading and physical object layout on disk are discussed in Section 4.5.

4 Query and Update Processing in Lore

As depicted in Figure 2, the basic steps that Lore follows when answering a query are: (1) the query is parsed; (2) the parse tree is preprocessed and translated into an OQL-like query; (3) a query plan is constructed; (4) query optimization occurs; and (5) the optimized query plan is executed. Query processing in Lorel is fairly conventional, with some notable exceptions:

- Because of the flexibility of Lorel, the preprocessing of the parse tree to produce the OQL-like query is complex. We have implemented the specification described in [AQM+96] and we will not discuss the issue further here.
- Although the Lore engine is built around standard operators (such as *Scan* and *Join*), some take an original flavor. For example, *Scan* may take as argument a general path expression, and therefore may entail complex searches in the database graph.
- A unique feature of Lore is its automatic coercion of atomic values. Coercion has an impact on the implementation of comparators (e.g., $=$ or $<$), but more importantly we shall see that it has important effects on indexing.

The result of a Lorel query is always a set of OEM objects, which become subobjects of a newly created *Result* object. The *Result* object is returned through the API. The application may then use routines provided by the API to traverse the result subobjects and display them in a suitable fashion to the user.

To illustrate the sequence of steps that Lore follows when answering a query, we will trace an example through query planning and then discuss the operators used to execute the query plan. Consider the query introduced in Section 2, whose OQL-like version is:

```
select O
from DBGroup.Member M, M.Office O
where exists A in M.Age : A > 30
```

The initial query plan generated for this query is given in Figure 3. Before discussing the various operators in this plan, it is necessary to first understand the flow of control and the auxiliary data structures used when executing such a plan.

4.1 Iterators and Object Assignments

Our query execution strategy is based on familiar database operators. We use a recursive *iterator* approach in query processing, as described in, e.g., [Gra93]. With iterators, execution begins at the top of the 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 operators, an iterator approach avoids creation of temporary relations.

The "tuples" we operate on are *Object Assignments*, or *OAs*. An OA is a simple data structure containing slots corresponding to range variables in the query, along with some additional slots depending on the form of the query. For example, the OA slots for the example query are shown in Figure 4. Intuitively, each slot within an OA will hold the oid of a vertex on a data path currently being considered by the query engine. For example, if OA1 holds the oid for member "Smith", then OA2 and OA3 can hold the oids for one of Smith's *Office* subobjects and one of his *Age* subobjects, respectively. Note that at a given point during query processing, not all slots of the current OA necessarily contain a valid oid. Indeed, the goal of query execution is to build complete OAs. Once a valid OA reaches the top of the query plan, oids in appropriate slots are used to construct a component of the query result.

4.2 Query Operators

We now briefly explain the query operators appearing as nodes in Figure 3; query operators not appearing in this plan are discussed later. Each operator takes a number of arguments, with the last argument being the OA slot that will contain the result of the operation. Exceptions to this are the *Select* and *Project* operators, which do not have a target slot.

The *Scan* operator, which is used in several leaf nodes, is similar in functionality to a relational scan. Here, however, instead of scanning the set of tuples in a relation, our scan returns all oids that are subobjects of a given object, following a specified path expression. The *Scan* operator is defined as:

```
Scan (StartingOASlot, Path_expression,
      TargetOASlot)
```

Scan starts from the oid stored in the `StartingOASlot`, and at each iteration places into the `TargetOASlot` the oid of the next subobject that satisfies the `Path_expression`, until there are no more matching subobjects. Note that in most cases `Path_expression` consists of a single label, however it may be a complex data structure representing an arbitrary component of a general path expression (recall Section 2.2), essentially a regular expression. For the regular expressions that we currently support, it is sufficient for the *Scan* operator to keep a run-time stack of objects visited in order to match the `Path_expression`. However, for general regular expressions a finite-state automaton is required. Recall that to avoid infinite numbers of matching paths, we match acyclic paths in the data only. Currently, the *Scan* operator can avoid traversing a cycle by ensuring that no oid appears more than once on its stack. Since the stack grows no larger than acyclic paths in the database, we do not expect its size to be a problem.

As a simple example of the *Scan* operator, consider the following node from our example plan:

```
Scan (OA1, "Office", OA2)
```

This iterator will place into slot OA2, one at a time, all *Office* subobjects of the object appearing in slot OA1. Note the special form for the lower left *Scan*:

```
Scan (Root, "DBGroup", OA0)
```

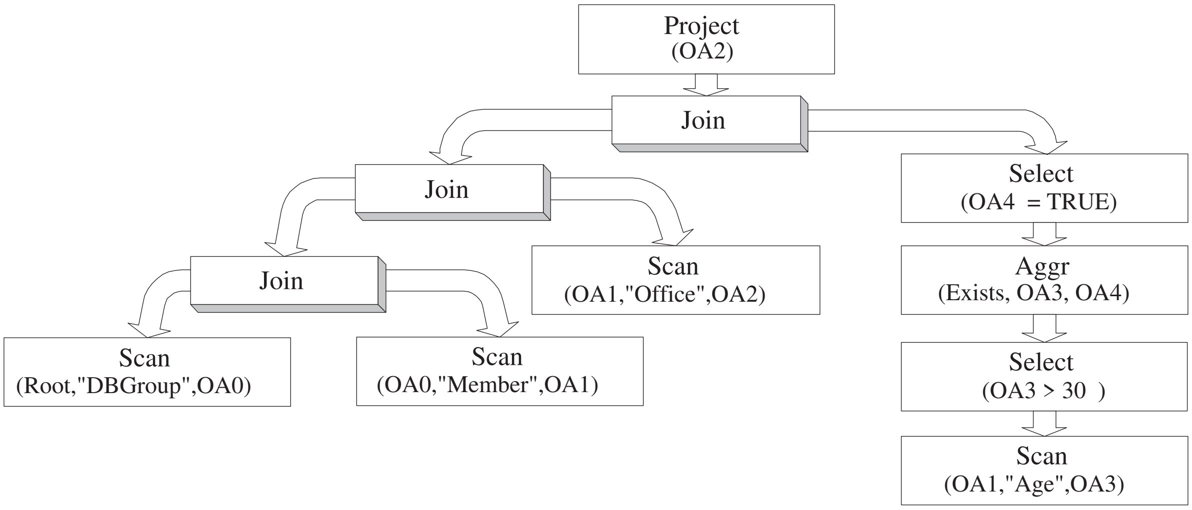

Figure 3: Example Lore query plan

OA0	OA1	OA2	OA3	OA4
(DBGroup)	(OA0.Member)	(OA1.Office)	(OA1.Age)	(true/false)

Figure 4: Example object assignment

Instead of using an OA slot as the first argument, the value *Root*, which is a system-known object from which all names (such as `DBGroup`) can be reached, is used.

The *Join*, *Project*, and *Select* nodes are nearly identical to their corresponding relational operators. Like a relational nested-loop join, the *Join* node coordinates its left and right children. For each partially completed OA that the left child returns, the right child is called exhaustively until no more new OAs are possible. Then the left child is instructed to retrieve its next (partial) OA. The iteration continues until the left side produces no more OAs. The *Project* node is used to limit which objects should be returned by specifying a set of OA slots, while the *Select* node applies a predicate to the object identified by the oid in the OA slot specified.

The *Aggregation* node (shown in Figure 3 on the right side of the query plan as `Aggr`) is used in a somewhat novel way, since it implements quantification as well as aggregation. At a high level, the aggregation node calls its child exhaustively, storing the results temporarily or computing the aggregate incrementally. When the child can produce no more valid OAs, a new object is created whose value is the final aggregation; this new object is identified within the target OA slot. In the example shown, the aggregation node adds to the target slot (OA4) the result of the aggregation, which here is the value *true* if the existential quantification is satisfied (an object exists in OA3) and *false* otherwise. Filtering of OAs whose quantification is *true* occurs in the *Select* node immediately above the aggregation node. Note that the *exists* aggregation operator "short circuits" when it finds the first satisfying OA, while other aggregation operators may need to look at all OAs.

There are four other primary query operators in Lore, in addition to operators for plans that use indexes (see Section 4.3): *SetOp*, *ArithOp*, *CreateSet*, and *Groupby*. *SetOp* handles the Lorel set operations *Union*, *Intersect*, and *Except*. Likewise, *ArithOp* handles arithmetic operations such as addition, multiplication, etc. *CreateSet* is used to package the results of an arbitrary subquery before proceeding; it calls its child exhaustively, storing each oid returned as part of a newly created complex object. After the child has produced all possible OAs, the *CreateSet* operator stores the oid for the new set of objects within the target slot in the OA. Finally, the *Groupby* operator handles (sub)queries that include a `groupby` expression.

To give a more in-depth flavor of query plan construction, we consider a second query. This query asks for the names and the number of publications for each database group member who is in the Computer Science ("CS") department.[3]

```
select M.Name, count(M.Publication)
from DBGroup.Member M
where M.Dept = "CS"
```

It is important to note that both `M.Name` and `M.Publication` appearing in the `select` clause are sets of objects, and in the general case are represented by subqueries. Thus, the OQL-like translation of this query is:

```
select (select N from M.Name N),
       count(select P
             from M.Publication P)
from DBGroup.Member M
where exists D in M.Dept : D = "CS"
```

To see the construction of the query plan, refer to Figure 5. The subtree for the `from` clause is constructed first. Each simple path expression (or range variable) appearing within the `from` becomes a *Scan* node. If several of these exist, then a left-deep tree of *Scan* nodes with *Join* nodes connecting them is constructed. At the top of the `from` subtree a *Join* node connects the `from` clause with the subtree for the `where` clause. For `where`, each `exists` becomes a *Select*, *Aggr*, and *Scan* node, and each predicate becomes a *Select* node. Finally, for the `select` clause, another *Join* node is added to the top of the tree, and the query plan subtree for the `select` clause becomes the right child.

Let us further consider the subtree for the `select` clause. The plans for the two expressions constituting the `select` clause are combined via union (using the *SetOp* operator).

[3] Several of our group members are in the Electrical Engineering department.

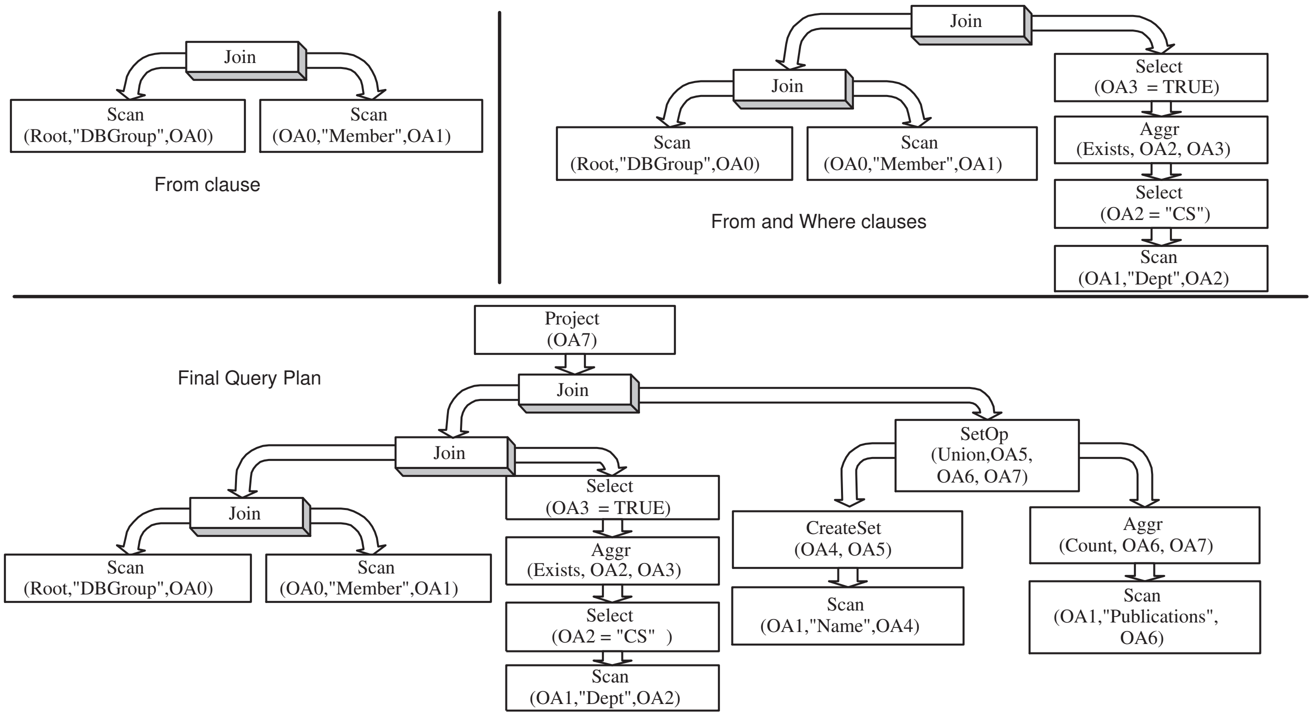

Figure 5: Steps in constructing a query plan

Thus, each (complex) object in the result contains the set of all *Name* subobjects of a *Member* (the left subtree of the *Union*), together with the count of all publications for that member. (In Lorel, a `select` list indicates union, while ordered pairs would be achieved using a tuple constructor operator [AQM+96].) The *CreateSet* operator, described earlier, is needed to obtain all *Name* children of a given member before returning its object assignment up the query tree. A *CreateSet* operator is not used in the right subtree, however, since the *Aggregation* operator by definition already calls its subquery to exhaustion (and then applies the aggregation operator, in this case `count`) before continuing.

4.3 Query Optimization and Indexing

The Lore query processor currently implements only a few simple heuristic query optimization techniques. For example, we do push selection operators down the query tree, and in some cases we eliminate or combine redundant operators. In the future, we plan to consider additional heuristic optimizations, as well as the possibility of truly exploring the search space of feasible plans.

Despite the lack of sophisticated query optimization, Lore does explore query plans that use indexes when feasible. 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. Thus, we have two kinds of indexes in Lore: a link (edge) index, or *Lindex*, and a value index, or *Vindex*. A Lindex takes an oid and a label, and returns the oids of all parents via the specified label. (If the label is omitted all parents are returned.) The Lindex essentially provides "parent pointers," since they are not supported by Lore's object manager. A Vindex takes a label, operator, and value. It returns all atomic objects having an incoming edge with the specified label and a value satisfying the specified operator and value (e.g., < 5). Because Vindexes

$arg1$ \ $arg2$	$string$	$real$	int
$string$	–	$string \rightarrow real$	$both \rightarrow real$
$real$	$string \rightarrow real$	–	$int \rightarrow real$
int	$both \rightarrow real$	$int \rightarrow real$	–

Table 1: Coercion for basic comparison operators

are useful for *range* (inequality) as well as *point* (equality) queries, they are implemented as B+-trees. Lindexes, on the other hand, are used for single object lookups and thus are implemented using linear hashing [Lit80].

Used in conjunction, these two kinds of indexes enable query processing in Lore to avoid the standard *Scan* operator. Before examining query plans that exploit indexes, we first take a more detailed look at Vindexes and how they handle the coercion present in Lorel.

4.3.1 Value Indexes

Value indexing in Lore requires some novel features due to its non-strict typing system. When comparing two values of different types, Lore always attempts to coerce the values into comparable types. Currently, our indexing system deals with coercions involving integers, reals, and strings only. Table 1 illustrates the coercion that Lore performs for these types; note that we simplify the situation by always coercing integers to reals. Now, in order to use Vindexes for comparisons, Lore must maintain three different kinds of Vindexes:

1. A *String Vindex*, which contains index entries for all string-based atomic values (`string`, `HTML`, `URL`, etc.).
2. A *Real Vindex*, which contains index entries for all numeric-based atomic values (`integer` and `real`).

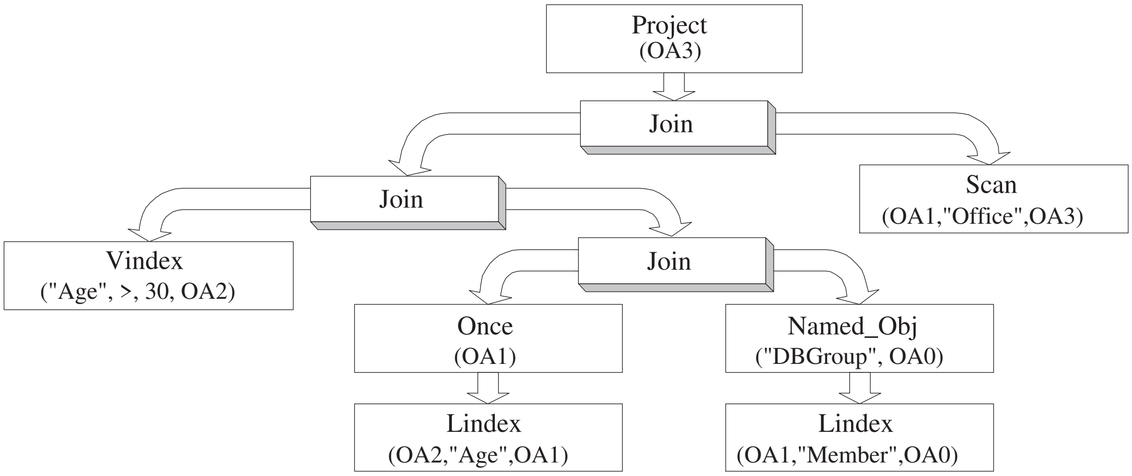

Figure 6: A query plan using indexes

3. A *String-coerced-to-real Vindex*, which contains all string values that can be coerced into an integer or real (stored as reals in the index).

For each label over which a Vindex is created, three separate B+-trees, one for each type, are constructed.

When using a Vindex for a comparison (e.g., find all *Age* objects > 30), there are two cases to consider, based upon the type of comparison value:

1. If the value is of type string, then: (i) do a lookup in the String Vindex; (ii) if the value can be coerced to a real, then also do a lookup for the coerced value in the Real Vindex.
2. If the value is of type real (or integer), then: (i) do a lookup in the Real Vindex; (ii) also do a lookup in the String-coerced-to-real Vindex.

4.3.2 Index Query Plans

If the user's query contains a comparison between a path expression and an integer, real, or string (e.g., "`DBGroup.Member.Age > 30`"), and the appropriate Vindexes and Lindexes exist, then a query plan that uses indexes will be generated. For simplicity, let us consider only queries in which the `where` clause consists of one such comparison.

Query plans using indexes are different in shape from those based on *Scan* operators. Intuitively, index plans traverse the database bottom-up, while scan-based plans perform a top-down traversal. An index query plan first locates all objects with desired values and appropriately labeled incoming edges via the Vindex. A sequence of Lindex operations then traverses up from these objects attempting to match the full path expression in the comparison.[4] Note that once we have an OA that satisfies the `where` clause, it may be necessary to use one or more *Scan* operations to find those components of the `select` expression that do not appear in the `where` clause.

Let us consider the following query (in its OQL-like form), first introduced in Section 2:

```
select O
from DBGroup.Member M, M.Office O
where exists A in M.Age : A > 30
```

A query plan using indexes is shown in Figure 6. This plan introduces four new query operators: *Vindex*, *Lindex*, *Once*, and *Named_Obj*. The *Vindex* operator, which appears as the left child of the second *Join* operator, iteratively finds all atomic objects with value less than 30 and an incoming edge labeled *Age*, placing their oids in slot OA2. The *Lindex* operator that appears below the *Once* operator iteratively places into OA1 all parents of the object in OA2 via an *Age* edge. (Since OEM data may have arbitrary graph structure, the object could potentially have several parents via *Age*, as well as parents via other labels.) Since *Age* is existentially quantified in the query, we only want to consider each parent once, even if it has several *Age* subobjects; this is the purpose of the *Once* query operator. The second *Lindex* operator finds all parents of the OA1 object via a *Member* edge, placing them in OA0. Since we want the object in OA0 to be the named object *DBGroup*, the *Named_Obj* operator checks whether this is so. Once we have traversed up the database using index calls and constructed a valid OA, we finally use a *Scan* operator to find all `Office` subobjects, which are returned as the result via the topmost *Project* operator.

Currently, for processing `where` clauses, Lore only considers subplans that are completely index-based (i.e., bottom-up), such as the one discussed here, or subplans that are completely *Scan*-based (i.e., top-down), such as the one in Figure 3. An interesting research topic that we have just begun to address is how to combine both bottom-up (index) and top-down (*Scan*) traversals. When the two traversals reach a predefined "meeting point", the intersection of the objects discovered by the index calls and the *Scan* operators identify paths that satisfy the `where` clause. The appropriate meeting point depends on the "fan-in" and "fan-out" of the vertices and labels in the database, and requires the use of statistical information.

4.4 Update Query Plans

Thanks to query plan modularity, we were able to handle arbitrary Lorel update statements by adding a single operator, *Update*, to the query execution engine. We illustrate the approach with our example update query from Section 2.2:

[4] An obvious alternative is to use full path indexes in place of the Lindex. Path indexes would be (much) more expensive to maintain but (much) faster at query time. Path indexes are discussed in more detail in [GW97].

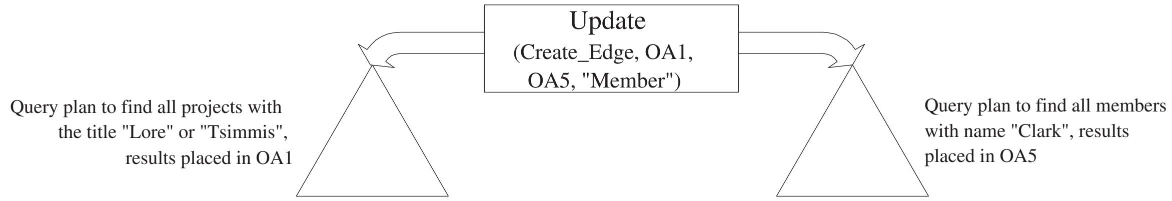

Figure 7: Example update query plan

```
update P.Member +=
   ( select DBGroup.Member
     where DBGroup.Member.Name = "Clark" )
from DBGroup.Project P
where P.Title = "Lore" or
      P.Title = "Tsimmis"
```

The query plan is outlined in Figure 7. The left subtree of the *Update node* computes the `from` and `where` clauses of the update. In our example, the left subtree finds those projects with title "Lore" or "Tsimmis". For each OA returned, the right subtree is called to evaluate the query plan for the subquery to the right of `+=`. (Other valid update assignment operators are `:=` and `-=` [AQM+96]). In our example, the right subtree finds those members whose name is "Clark". Once the right subtree completes the OA, the *Update* node performs the actual update operation; valid operations are *Create_Edge*, *Destroy_Edge*, and *Modify_Atomic*. In our example, the *Update* node creates an edge labeled *Member* between each pair of objects identified by its subtrees. Clearly a number of optimizations are possible in update processing. For instance, in our example the right subtree of the *Update* node is uncorrelated with the left subtree and thus needs to be executed only once. We currently perform this optimization, and we are investigating others.

4.5 Bulk Loading and Physical Storage

Data can be added to a Lore database in two ways. Either the user can issue a sequence of update statements to add objects and create labeled edges between them, or a load file can be used. In the latter case, a textual description of an OEM database is accepted by a load utility, which includes useful features such as symbolic references for shared subobjects and cyclic data, as well as the ability to incorporate new data into an existing database.

Lore arranges objects in physical disk pages; each page has a number of slots with a single object in each slot. Since objects are variable-length, Lore places objects according to a *first-fit* algorithm, and provides an object-forwarding mechanism to handle objects that grow too large for their page. In addition, Lore supports large objects that may span many pages; such large objects are useful for our multimedia types, as well as for complex objects with very broad fan-out. Objects are clustered on a page in a depth-first manner, primarily because our *Scan*-based plans traverse the database depth-first. It is obviously not always possible to keep all objects close to their parents since an object may have several parents. For now, if an object has multiple parents then it is stored with an arbitrary parent. Finally, if an object o cannot be reached via a path originating from a named object, then o is deleted by our garbage collector.

5 Novel Features

This section provides brief overviews of two novel features of Lore: the external data manager and DataGuides. Due to space constraints, coverage is cursory, but should give the reader a flavor of these components. For further details on the external data manager see [MW97]. Further details on DataGuides can be found in [GW97].

5.1 External Data

Lore's *external data manager* enables dynamic retrieval of information from other data sources based on queries issued to Lore. The externally obtained data is combined with resident Lore data during query evaluation, and the distinction between the two types of data is invisible to the user. (Thus, external data in Lore provides a way to query distributed information sources by essentially transforming Lore into an information integration engine.) An *external object* stored within a Lore database functions as both a placeholder for the external data, and specifies how Lore interacts with the external data source. During query processing, when the execution engine discovers an external object, information is fetched from the external source to answer the query, and the fetched information is cached within the Lore database until it becomes "stale."

Clearly there are many possible approaches that can be taken to integrate external data in this fashion. Our main motivation in choosing the approach outlined below was to enable Lore to bring in data from a wide variety of external sources, and to introduce a variety of argument types and optimization techniques to limit the amount of data fetched from an external source to that which is immediately useful in answering a given query. Because the related *Tsimmis* project at Stanford has focused on building "wrappers" that provide OEM interfaces to arbitrary data sources [PGGMU95], we are able to easily exploit such sources as external data in Lore.

In Figure 8, we see the *logical* and *physical* views of a small database with an external object (shaded in the figure). The logical view is that seen by the user, as if the external data is stored in Lore. The physical view shows how Lore encodes the information associated with an external source, along with any fetched data. The sample database contains information about member "Jim", where Jim's publication information is obtained externally. During query processing, the *Scan* operator notifies the external data manager whenever an external object is encountered. The external data manager may need to fetch information from the external source, and will provide back to the *Scan* operator zero or more oids that are used in place of the oid of the external object. Query processing then proceeds as normal.

The physical view in Figure 8, simplified from the actual implementation, shows that the specification for an external object includes: (i) the location of a *Wrapper* program that fetches the external data and translates it into OEM, (ii) a *Quantum* that indicates the time interval until fetched information becomes stale, and (iii) a set of *Arguments* that are used to limit the information fetched in a call to the external source. Arguments sent to the external source can come from three places: the query being processed (*query-defined*), values of other objects in the local database (*data-defined*), or constant values tied to the exter-

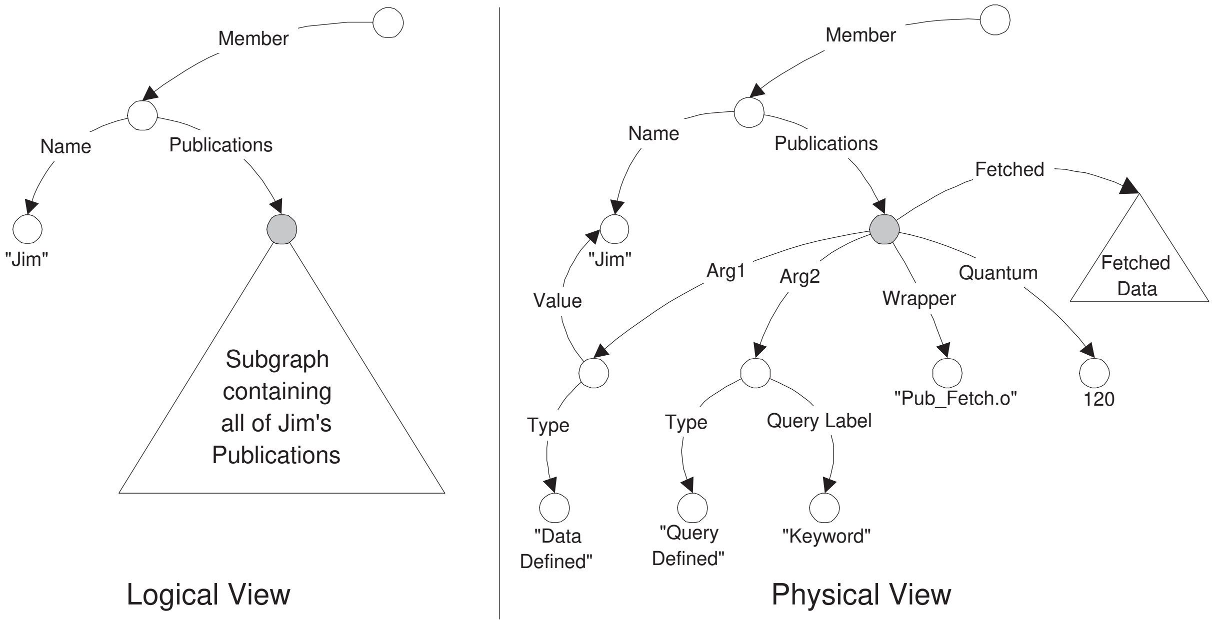

Figure 8: The logical and physical views of the data

nal object (*hard-coded*). Example data-defined and query-defined arguments can be seen in Figure 8 as *Arg1* and *Arg2* respectively. The value of the atomic object pointed to by the *Value* edge from *Arg1* is sent to the data source as one argument. In the query-defined argument specification, the *Query Label* object with value "Keyword" specifies that if the query being processed has a predicate of the form "`Member.Publications.Keyword = X`", then X is sent to the external data source as another argument.

Many calls to an external source can quickly dominate query processing time. We briefly mention two of the ways our external data manager attempts to limit the number of calls. First, if a single query will result in multiple calls to an external source (due to multiple bindings for data-defined and/or query-defined arguments), then we have a mechanism for recognizing when a call to an external source will subsume another scheduled call with a different argument set, and we eliminate the second call. Second, we track the argument sets used by previous queries and determine when previously fetched (non-stale) information partially or entirely subsumes information required by the current argument set. A more detailed description of argument sets and optimizations appears in [MW97].

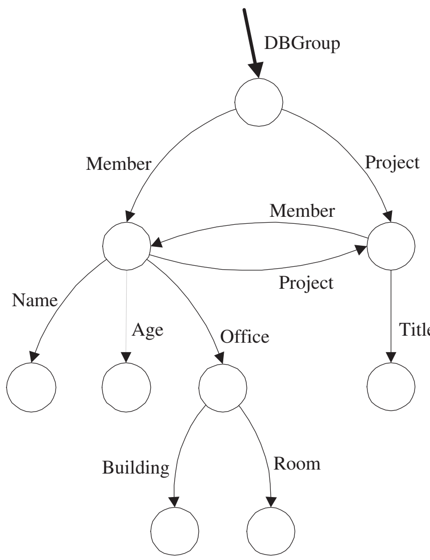

Figure 9: A DataGuide for Figure 1

5.2 DataGuides

Since a Lore database does not have an explicit schema, query formulation and query optimization are particularly challenging. Without some knowledge of the structure of the underlying database, writing a meaningful Lorel query may be difficult, even when using general path expressions. One may manually browse a database to learn more about its structure, but this approach is unreasonable for very large databases. Further, without information about the structure of the database, the query processor may be forced to perform more work than necessary. For example, consider the query plan discussed in Section 4, which finds the offices of all group members older than 30. Even if no members have an office, the query plan would needlessly examine every member in the database.

A *DataGuide* is a concise and accurate summary of the structure of an OEM database, stored itself as an OEM object. Each possible path expression of a database is encoded exactly once in the DataGuide, and the DataGuide has no path expressions that do not exist in the database. In typical situations, the DataGuide is significantly smaller than the original database. Figure 9 shows a DataGuide for the sample OEM database from Figure 1. In Lore, a DataGuide plays a role similar to metadata in traditional database systems. The DataGuide may be queried or browsed, enabling user interfaces or client applications to examine the structure of the database. As will be seen in the next section, an interactive DataGuide is an important part of Lore's Web interface. Assuming the role of the missing schema, the DataGuide can also guide the query processor. Of course, in relational or object-oriented systems the schema is explicitly created before any data is loaded; in Lore, DataGuides are dynamically generated and maintained over all or part of an existing database.

For a given OEM database, there are many DataGuides that satisfy the desired properties specified above (accuracy and conciseness). For example, in Figure 9 we could fuse all leaf objects into a single object without changing the fact that every path expression is encoded exactly once (and without adding superfluous paths). It turns out that certain DataGuides are much easier to keep consistent in response to updates to the underlying database. In addition, some DataGuides support storage of *annotations* within objects: properties of the set of objects reachable by a path expression in the original database. We store an annotation for a given path expression by assigning it to the single object in the DataGuide reachable by that path expression. Annotations are useful, e.g., for storing sample atomic values reachable via a given path expression, or for specifying the statistical chances of finding an outgoing edge with a certain label.

In [GW97], formal definitions for DataGuides are provided as well as algorithms to build and incrementally maintain DataGuides that support annotations. Also given is a discussion of how DataGuides aid query formulation in practice and their use for query optimization.

6 Interfaces to Lore

As shown in Figure 2, the Lore Application Programming Interface (API) provides a gateway between Lore and any user interface or client application. It is used, for instance, by the system's textual interface, which passes user commands to Lore and presents query results in a hierarchical display. After summarizing the API, we describe a Java-based Web interface that makes Lore simple to use in an interactive fashion.

6.1 Application Programming Interface

The Lore API is composed of a small collection of C++ classes. For any client, Lore is simply viewed as a single library, accessible through the API classes and methods declared in a single header file. (Eventually we hope to move Lore toward a traditional client-server model.) At the highest level, the API allows a client program to connect to a Lore database, submit queries and commands, and process query results.

Any session with a Lore database is encapsulated in an instance of the `LoreConnection` class. A client will first `Connect` to a specific database (and eventually `Disconnect` when finished). Clients submit Lorel queries using the `Submit` function. `Submit` is also used for other Lore system commands, such as index creation and updates. When called with a Lorel query, `Submit` returns the query result as a `LoreOem` object. A `LoreOem` instance initially contains only an oid; the actual value is fetched from the database on demand. For atomic objects, a client may request the `Type` and `Value` of the object. To traverse the subobjects of a complex object, a client instantiates a `LoreIterator`. Each successive call to the iterator's `Next` method returns a different `LoreOem` subobject and its `Label`. By nesting `LoreIterator` instances, a client may perform arbitrary traversals of OEM objects.

6.2 Web Interface

A user connects to our graphical Web interface by visiting a specific URL and choosing a database. The user is

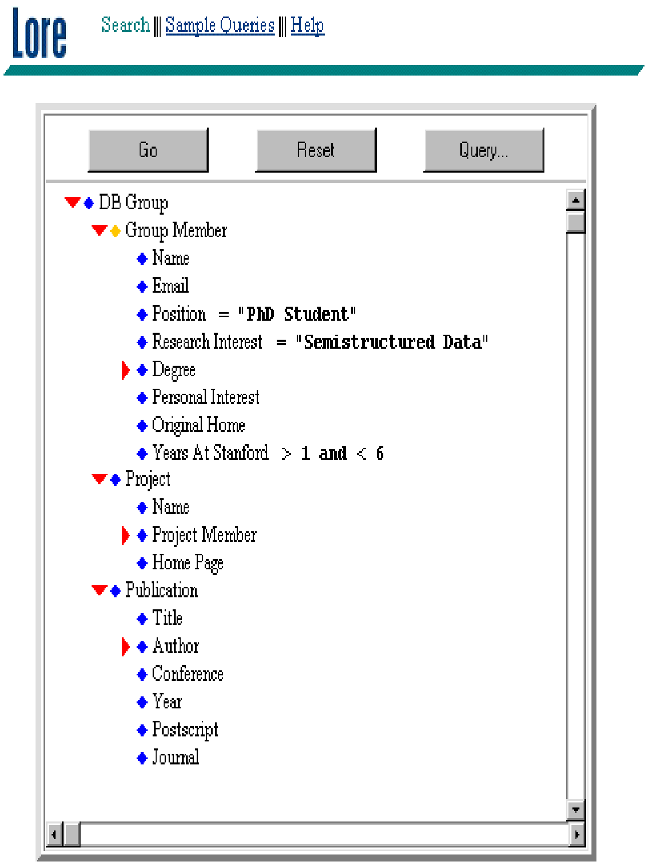

Figure 10: A DataGuide in Java

then presented with a Java program featuring a DataGuide, as described in Section 5.2. Users can quickly and easily browse the DataGuide to explore the structure of the underlying database. Through the Web interface, the user may submit a textual Lorel query or select a sample prewritten query. Furthermore, in a style similar to *Query-By-Example* [Zlo77], queries may be formulated and submitted without any knowledge of Lorel by using the DataGuide to select path expressions and specify selection conditions. Currently, DataGuide queries can express Lorel queries with simple path expressions and a `where` clause that is conjunctive with respect to unique path expressions.

As an example, Figure 10 is a screen snapshot of the Java presentation of a DataGuide. This DataGuide summarizes an existing database for Stanford's Database Group, similar in structure to (but much larger than) the sample database used throughout this paper. Arrows accompany complex objects and are used to expand or collapse subobjects. Also, a diamond is associated with each displayed label, corresponding to a unique path expression from the root. When the user clicks on a diamond, a dialog box pops up, from which the user may view sample values, select the path expression for the query result, or add filtering conditions. When the user selects a path expression, the corresponding diamond is rendered in a different color. Filtering conditions are displayed next to the corresponding label. The DataGuide shown in Figure 10 represents a query to select all group members that are PhD students, have a research interest in semistructured data, and have been at Stanford more than one year but less than six. When the user clicks *Go*, the Java program automatically generates an equivalent Lorel query and sends it to Lore to be processed.

Regardless of how a query is submitted, the interface displays query results in HTML, in a hierarchical format that is easy to read and navigate. By formatting OEM objects in HTML, we can leverage Web browser support for our multimedia data (such as `gif` files, audio, or video). To make the hierarchical display of OEM more readable, we perform two small presentation transformations. First, if several objects share the same label, we display the label only once and show the values of the objects underneath it. For example, if a query result contains ten objects, each with the same label `Project`, we create an HTML page that begins with a single header `Projects`, followed by the values for all ten projects. Second, we present complex OEM objects as active hyperlinks. Clicking on the link brings up a new HTML page showing the subobjects of that complex object.

7 System Status and Future Work

As of June 1997, the Lore system is functional and robust for a large subset of the Lorel language. It consists of approximately 60,000 lines of C++ code. Some language features, such as external predicates and functions, are still under implementation. Also, general path expressions are not yet implemented in their full generality, although a substantial and very useful subset is.

A Lore server with sample databases is available for public use. Users can submit queries and can experiment with features such as DataGuides and result browsing. To visit our on-line demo, see `http://www-db.stanford.edu/lore`. In addition, Lore system binaries for several platforms are available through the Web page.

We are considering many possible enhancements and extensions to Lore, as follows.

7.1 Compatibility and Interoperability

As mentioned in Section 2 and covered in detail in [AQM+96], OEM and Lorel can be translated to ODMG and OQL [Cat94]. In the translation, OEM objects are represented by ODMG objects, while Lorel queries are transformed into pure OQL queries that use method calls to handle Lorel features such as type coercion and general path expressions. As a proof-of-concept for the translation, we have implementing Lorel on top of the O_2 object-oriented database management system [BDK92]. Note that this implementation enables the storage of semistructured (OEM) and structured (ODMG) data in a single repository, providing a useful setting in which we are studying integration of the two data models. We also plan to explore how Lorel could be translated to SQL3 and thus implemented on top of an object-relational database management system.

7.2 Performance Issues

To date we have done little performance analysis of Lore. There are a number of performance aspects we want to consider, such as overall performance and bottlenecks in the system, scalability of the system to extremely large databases, and comparing the performance of Lore against our implementation of Lorel on top of O_2 (see Section 7.1).

There is significant additional research to do in query optimization, including query rewriting, operation ordering, selecting the best use of indexes in query plans, and exploiting information stored in the DataGuide.

As described in Section 4.3, we can build in Lore a *link index* (*Lindex*) in order to quickly find all parents of a given object reachable via a given label. Alternatively, we could instead augment our storage manager to store with objects their inverse (parent) pointers in addition to their subobject (child) pointers. We plan to compare the performance of a storage manager with inverse pointers to that of our current approach based on Lindexes. We also plan to consider using *path indexes* in place of the Lindex. Interestingly, the functionality of path indexes is incorporated easily into the DataGuide, as discussed in [GW97].

Currently all "expansions" of path expressions in query paths are done at run-time. However, for some classes of path expressions, it is possible to use information in the DataGuide to expand the regular expressions to all possible completions at query compilation time. We plan to explore the compile-time approach and compare its performance against the run-time approach we now take.

7.3 New Functionality

We are in the process of implementing transaction support for concurrency control and recovery. As with other aspects of Lore, the semistructured nature of Lore's data is requiring us to rethink some aspects of traditional solutions.

In the user interface area, we plan to increase the expressiveness of DataGuide queries toward the full power of Lorel. In addition, to follow the recent trend of enabling database systems to dynamically generate customized HTML displays of query results [Gaf97, BDK92], we plan to investigate more sophisticated techniques for customizing the presentation of OEM objects in a Web environment.

In a companion project, we have extended OEM and Lorel in order to treat changes to the data as a first-class concept [CAW97], similar to the *Heraclitus* system that operates on structured data [GHJ96]. Currently we are implementing this model and language on top of the Lore system.

Initial work is underway to define both view and trigger mechanisms appropriate for semistructured data, and to implement them in Lore. (See [AGM+97] for a discussion of views in the context of OEM and Lorel.) Finally, because many applications appropriate for a semistructured DBMS such as Lore include a significant amount of text data, we plan to incorporate a special `text` type along with a full-text indexing system into Lore.

Acknowledgments

For their many contributions to the Lore project and system implementation we are grateful to (alphabetically) Kevin Haas, Matt Jacobsen, Tirthankar Lahiri, Qingshan Luo, Svetlozar Nestorov, Anand Rajaraman, Hugo Rivero, Michael Rys, and Takeshi Yokokawa. We also thank many other members of the Stanford Database Group for fruitful discussions about Lore and Lorel, including (alphabetically) Sudarshan Chawathe, Joachim Hammer, Shuky Sagiv, Jeff Ullman, Janet Wiener, and Jun Yang. Finally, we are grateful to an anonymous referee for a careful reading and helpful comments.

References

[Abi97] S. Abiteboul. Querying semistructured data. In *Proceedings of the International Conference on Database Theory*, 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.

[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), November 1996.

[BCK+94] G. Blake, M. Consens, P. Kilpeläinen, P. Larson, T. Snider, and F. Tompa. Text/relational database management systems: Harmonizing SQL and SGML. In *Proceedings of the First International Conference on Applications of Databases*, pages 267–280, Vadstena, Sweden, 1994.

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

[BDK92] F. Bancilhon, C. Delobel, and P. Kanellakis, editors. *Building an Object-Oriented Database System: The Story of O_2*. Morgan Kaufmann, San Francisco, California, 1992.

[BDS95] P. Buneman, S. Davidson, and D. Suciu. Programming constructs for unstructured data. In *Proceedings of the 1995 International Workshop on Database Programming Languages (DBPL)*, 1995.

[BK94] C. Beeri and Y. Kornatski. A logical query language for hypermedia systems. *Information Sciences*, 77, 1994.

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

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

[CAW97] S. Chawathe, S. Abiteboul, and J. Widom. Representing and querying changes in semistructured data. Technical report, Stanford University Database Group, February 1997.

[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.

[CM89] M.P. Consens and A.O. Mendelzon. Expressing structural hypertext queries in GraphLog. In *Proceedings of the Second ACM Conference on Hypertext*, pages 269–292, Pittsburgh, Pennsylvania, November 1989.

[Com91] IEEE Computer. *Special Issue on Heterogeneous Distributed Database Systems*, 24(12), December 1991.

[Gaf97] J. Gaffney. Illustra's web datablade module. Technical report, Informix Corporation, February 1997. Available at `http://www.informix.com` as `/informix/corpinfo/zines/whitpprs/illuswp/dblade.htm`.

[GHJ96] S. Ghandeharizadeh, R. Hull, and D. Jacobs. Heraclitus: Elevating deltas to be first class citizens in a database programming language. *ACM Transactions on Database Systems*, 21(3):370–426, 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*, Athens, Greece, August 1997.

[KKS92] M. Kifer, W. Kim, and Y. Sagiv. Querying object-oriented databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 393–402, San Diego, California, June 1992.

[KS95] D. Konopnicki and O. Shmueli. W3QS: A query system for the World Wide Web. In *Proceedings of the Twenty-First International Conference on Very Large Data Bases*, pages 54–65, Zurich, Switzerland, September 1995.

[Lit80] W. Litwin. Linear hashing: a new tool for file and table addressing. In *Proceedings of the International Conference on Very Large Data Bases*, pages 212–223, Montreal, Canada, October 1980.

[LMR90] W. Litwin, L. Mark, and N. Roussopoulos. Interoperability of multiple autonomous databases. *ACM Computing Surveys*, 22(3):267–293, 1990.

[LSS96] L.V.S. Lakshmanan, F. Sadri, and I.N. Subramanian. A declarative language for querying and restructuring the Web. In *Proceedings of the Sixth International Workshop on Research Issues in Data Engineering (RIDE '96)*, New Orleans, February 1996.

[MMM96] A.O. Mendelzon, G. Mihaila, and T. Milo. Querying the world wide web. In *Proceedings of the Conference on Parallel and Distributed Information Systems (PDIS'96)*, December 1996. Full version to appear in the Journal of Digital Libraries.

[MS93] J. Melton and A.R. Simon. *Understanding the New SQL: A Complete Guide*. Morgan Kaufmann, San Francisco, California, 1993.

[MW93] T. Minohara and R. Watanabe. Queries on structure in hypertext. In *Proceedings of the Conference on Foundations of Data Organization (FODO '93)*, pages 394–411. Springer Verlag, 1993.

[MW95] A. O. Mendelzon and P. T. Wood. Finding regular simple paths in graph databases. *SIAM Journal of Computing*, 24(6), 1995.

[MW97] J. McHugh and J. Widom. Integrating dynamically-fetched external information into a dbms for semistructured data. In *Proceedings of the Workshop on Management of Semistructured Data*, pages 75–82, Tucson, Arizona, May 1997.

[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.

[PGGMU95] Y. Papakonstantinou, A. Gupta, H. Garcia-Molina, and J. Ullman. A query translation scheme for rapid implementation of wrappers. In *Proceedings of the Fourth International Conference on Deductive and Object-Oriented Databases*, Singapore, December 1995.

[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.

[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.

[QWG+96] D. Quass, J. Widom, R. Goldman, K. Haas, Q. Luo, J. McHugh, S. Nestorov, A. Rajaraman, H. Rivero, S. Abiteboul, J. Ullman, and J. Wiener. LORE: A Lightweight Object REpository for Semistructured Data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, page 549, Montreal, Canada, June 1996. Demonstration description.

[SK91] M. Stonebraker and G. Kemnitz. The POSTGRES next-generation database management system. *Communications of the ACM*, 34(10):78–92, October 1991.

[SL90] A. Sheth and J.A. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Computing Surveys*, 22(3):183–236, 1990.

[YA94] T. Yan and J. Annevelink. Integrating a structured-text retrieval system with an object-oriented database system. In *Proceedings of the Twentieth International Conference on Very Large Data Bases*, pages 740–749, Santiago, Chile, September 1994.

[Zlo77] M.M. Zloof. Qurey-by-Example: a data base language. *IBM Systems Journal*, 16(4):324–343, 1977.