Indexing Semistructured Data*

Jason McHugh Jennifer Widom Serge Abiteboul Qingshan Luo Anand Rajaraman
Computer Science Department
Stanford University
Stanford, CA 94305
Firstname.Lastname@cs.stanford.edu
http://www-db.stanford.edu/lore

Abstract

This paper describes techniques for building and exploiting indexes on *semistructured* data: data that may not have a fixed schema and that may be irregular or incomplete. We first present a general framework for indexing values in the presence of automatic type coercion. Then based on *Lore*, a DBMS for semistructured data, we introduce four types of indexes and illustrate how they are used during query processing. Our techniques and indexing structures are fully implemented and integrated into the Lore prototype.

1 Introduction

We call data that is irregular or that exhibits type and structural heterogeneity *semistructured*, since it may not conform to a rigid, predefined schema. Such data arises frequently on the Web, or when integrating information from heterogeneous sources. In general, semistructured data can be neither stored nor queried in relational or object-oriented database management systems easily and efficiently. We are developing *Lore*[1], a database management system designed specifically for semistructured data [MAG+97]. Building Lore from scratch allows us to explore how semistructured data affects each component of a database management system. In this paper we focus on indexing.

In any DBMS, the tradeoff between efficient query performance versus space and update cost must be considered. Indexing allows fast access to data by essentially replicating portions of the database in special-purpose structures. However, these structures must be kept up to date incrementally: each change to the base data must be reflected in all applicable indexes. Despite the cost of index maintenance, the added storage, and the added complexity in the query engine, indexes have shown themselves to be a useful and integral part of all database systems. The considerations involving the use of indexes are very similar in a database system for semistructured data: The administrator may choose to build some indexes which, if exploited properly by the query processor, will decrease query processing time. Consistency between the index structures and the data must be maintained. Many applications of semistructured data (such as querying Web or integrated data) tend to be

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

[1] Lore, for Lightweight Object REpository, has grown from a "lightweight" system for "lightweight" objects into a full-featured DBMS.

read-intensive, in which case the balance falls towards maintaining extensive indexing structures to speed up query processing. For Lore applications we often find this to be the approach of choice.

Semistructured data does introduce some novel problems concerning the actual mechanics of indexing. First, indexes in relational and object-oriented database systems are created over a set of attributes for some collection where the types of the attributes are defined in advance. In Lore, the data is essentially an arbitrary labeled directed graph [PGMW95], so it is difficult to isolate an attribute of a collection to index, and the type of objects are not known in advance. Second, Lore automatically performs type coercion when comparing objects of different types—an essential feature when dealing with semistructured data. Traditional value indexing techniques must be modified considerably in order to be used in the presence of coercion.

Indexing atomic values in our graph-based data model allows the query engine to quickly locate specific leaf objects. However, since almost all queries also explore the data via labeled traversals through the graph, we introduce additional indexes that efficiently locate edges and paths through the data. Lore also includes a simple full-text indexing system that efficiently supports Information Retrieval style predicates within Lore's query language. Finally, we describe how Lore's query optimizer pieces together the available indexes to create efficient query plans. All structures, index creation algorithms, index maintenance, and query plans described in this paper are implemented and fully functional in Lore.

1.1 Related Work

A description of the Lore architecture and query execution engine can be found in [MAG+97]. Lore's cost-based query optimizer was introduced in [MW97] with a focus on how the optimizer finds efficient query plans. Lore *DataGuides* [GW97], which are dynamic structural summaries of a database, serve both as a tool for the end-user and as a simple path index. A preliminary version of Lore's query language, *Lorel*, was introduced in [QRS+95]. Details of the current version of Lorel appear in [AQM+97].

Needless to say, there has been a significant amount of work in indexing for object-oriented databases, e.g., [KKD89, SS94, RK95, KM92, CCY94, BG92, XH94]. All of this work depends on the database having a fixed schema based on a known, strongly-typed class hierarchy. In our environment we must take a different approach to indexing, since we do not have a fixed schema, and comparable objects may take on different types.

Other related work includes recent research into efficient evaluation of *generalized path expressions*. In [CCM96], an algebraic framework and transformations are presented to optimize evaluation of generalized path expressions, but the approach does not consider the use of indexes. In [FS98], a notion of "state extent" is introduced for path expression evaluation that is similar in spirit to how we use DataGuide path indexes. However, [FS98] does not consider explicit storage or maintenance of state extents, nor how they are integrated with query plan generation, so they cannot be considered indexes in the traditional sense.

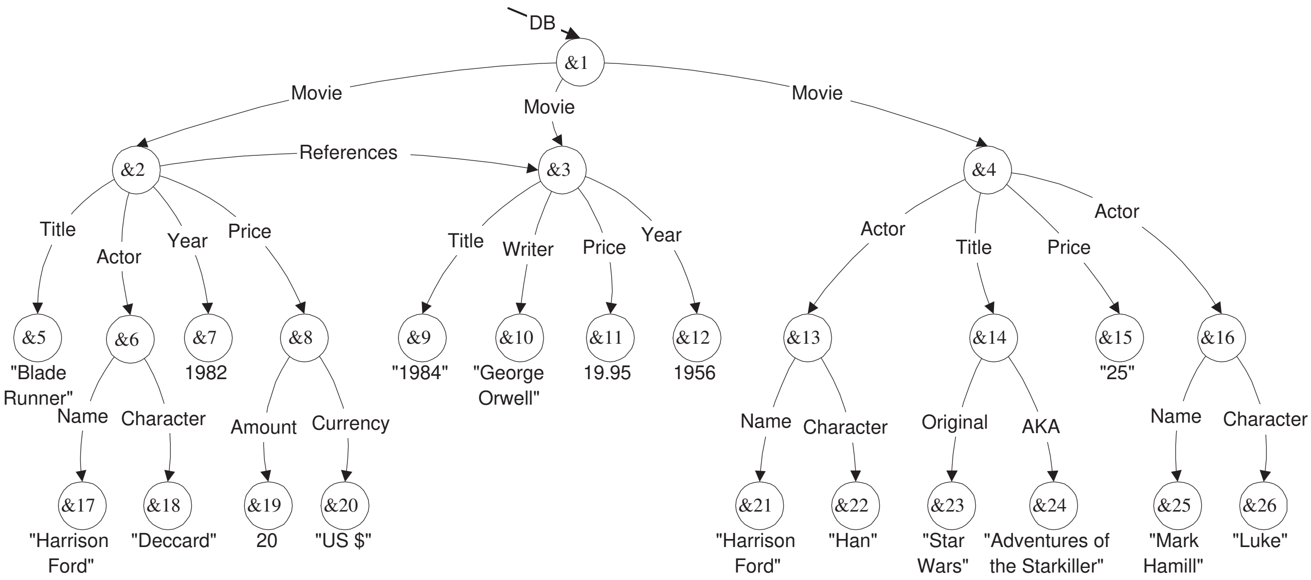

Figure 1: An OEM database

1.2 Paper Outline

Section 2 provides a brief introduction to the data model and query language used in Lore and motivates Lore's query execution strategies. A detailed foundation for indexing values in the presence of automatic type coercion, which is instantiated in Lore, is given in Section 3. The four index types implemented in the Lore system are described in Section 4. Query operators and plans that use the index structures are described in Section 5.

2 Preliminaries and Motivation

To set the stage for our discussion of indexing semistructured data, we first introduce Lore's data model, query language, and query execution strategies. For further details see [AQM+97, MW97].

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 OEM graph shown in Figure 1 depicts a tiny portion of a database containing information about movies. (Although the example database is mostly tree-structured, OEM and Lore permit arbitrary graph-structured databases.) 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 &4 is complex and its *subobjects* are &13, &14, &15, and &16.

Object &5 is atomic and has value "Blade Runner". *Names* are special labels that serve as aliases for objects and as entry points into the database. In Figure 1, *DB* is a name that denotes object &1.

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) movies have zero, one, or more actors; (ii) a price may be a string, a real, or even a complex object.

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 Lore's query language, described next.

2.2 The Lorel Query Language

We introduce the *Lorel* query language with a few examples. A full specification of Lorel, which is an extension of OQL, appears in [AQM+97]. Here we highlight those features of the language that are designed specifically for querying semistructured data and are relevant to our indexing scheme. Many other features of Lorel (some inherited from OQL and others not) 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, `DB.Movie.Actor` is a simple path expression. It denotes the set of objects that can be reached starting with the *DB* object, following an edge labeled *Movie*, then following an edge labeled *Actor*. Range variables can be assigned to path expressions, e.g., "`DB.Movie X`" specifies that X ranges over the movie subobjects of the name *DB*.

Our first example gives a simple Lorel query which, when executed over the database in Figure 1, returns the titles of all movies that Harrison Ford acted in. In the query result, indentation is used to represent graph structure.

Example 2.1

select DB.Movie.Title
where DB.Movie.Actor.Name = "Harrison Ford"

```
RESULT
   Title "Blade Runner"
   Title
      Original "Star Wars"
      AKA "Adventures of the Starkiller"
```

The database in Figure 1 presents a number of irregularities, even with respect to this simple query. A guiding principle in Lorel is that to write a query one should not have to worry about such irregularities (e.g., not all movies have actors), or know the precise structure of objects (e.g., the structure of *Titles*), nor should one have to bother with precise types. As we will show, automatic coercion between atomic values with different types complicates the use of value indexes.

The Lore query processor rewrites all queries into an OQL style. Our next example gives the translation of Example 2.1 into the equivalent OQL-like query. The user is free to enter either form of the query. The Lore system always builds its query execution strategy based upon the translated query.

Example 2.2

select T
from DB.Movie M, M.Title T
where exists A in M.Actor : exists N in A.Name : N = "Harrison Ford"

Lorel, like *Postquel* [SK91], allows the user to omit the `from` clause, as we have done in Example 2.1. Notice that in the rewritten version of the query a `from` clause has been introduced. Also notice that the comparison on the subpath `Actor.Name` has been transformed into two existential conditions. Thus, the user can write `DB.Movie.Actor.Name = "Harrison Ford"` regardless of whether `Actor.Name` is known to be single-valued, known to be set-valued, or unknown.

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. The use of a path pattern is shown in our next example.

Example 2.3

select DB.Movie.Price(.Amount)?
where DB.Movie.Year < DB.Movie.Title

```
RESULT
   Price 19.95
```

A "?" indicates that a subpath (in parentheses) is optional, so the path expression in the `select` clause will match both `DB.Movie.Price` and `DB.Movie.Price.Amount`. The query will return all *Price* and *Amount* subobjects for every movie where its year of production is less than its title: i.e., movies whose titles refer to future events, such as the movie "1984" which was produced in 1956. According to Lorel semantics, the comparison in the `where` clause returns false unless the values being compared can be coerced into the same atomic type [AQM+97]. In our example database, only &9 and &12 can be coerced into comparable types, and the comparison returns true since 1956 < 1984. Thus, object &3 satisfies the query and its `Price` subobject is returned. Further details and a formalization of atomic type coercion as implemented in Lore are given in Section 3.

2.3 Query Execution Strategies

Now let us consider some possible query execution strategies for Example 2.2, repeated here for convenience:

> **select** T
> **from** DB.Movie M, M.Title T
> **where** exists A in M.Actor : exists N in A.Name : N = "Harrison Ford"

The most straightforward approach is to first consider the `from` clause, finding objects along the path `DB.Movie.Title`. Then, for each *Movie*, we look for the existence of a path `Actor.Name` whose value is "Harrison Ford". We call this a *top-down* execution strategy since we begin at the named object *DB* (the top) and evaluate the path expressions in a forward manner. This query execution strategy results in a depth-first traversal of the graph and does not require any indexes.

Another way to execute the same query is to first identify all objects satisfying the `where` clause by using an index structure that locates all atomic objects with the value "Harrison Ford". Once we have an object satisfying the predicate, we traverse backwards through the data, matching path expressions in reverse. Since Lore's object store does not support parent (inverse) pointers, we must use an additional index structure to locate parents of a given object. 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. Obviously, whether bottom-up outperforms top-down depends on the query and the data, but we have found bottom-up to be a good strategy in practice when it is applicable.

A third strategy 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 we directly identify those atomic objects that satisfy the `where` using an indexing structure, and using another index we traverse up to the same point as the top-down exploration. By intersecting the sets of "valid" objects and combining traversed paths we obtain the result of the query. We call this approach a *hybrid* plan, since it operates both top-down and bottom-up, meeting in the middle of a path expression.

Yet another query execution strategy works *inside-out* by identifying portions of the graph that match the middle of a path expression, then traversing up through the parents and down through the children to match the remaining portions of the path expression. Here an index is needed to identify objects that match the middle of a path expression, as well as an additional index for the upward traversal.

These execution strategies give the reader a flavor of query processing and the use of indexes in Lore. We will return in more detail to Lore's query plans after we cover the indexing of coercible values and describe in more concrete terms the various indexes available in Lore.

3 Indexing Coercible Values

Automatically coercing values of different types for comparisons is an important feature in semi-structured data, when the user may not know the types of atomic data elements, or when the types of related atomic data may vary. In Example 2.3 it made sense to attempt to coerce the movie name

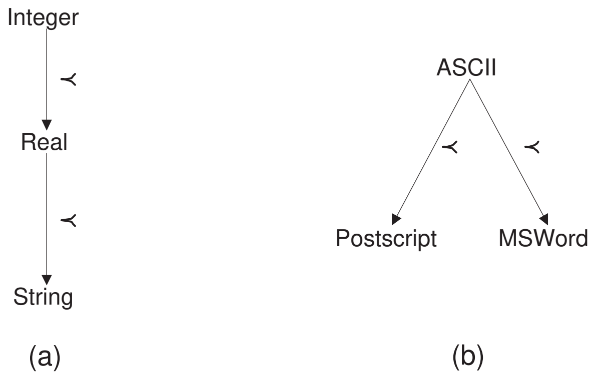

Figure 2: Partial orders among types

to an integer so it could be compared with the production year. Any query comparing the movie prices in our database will require coercion since the prices are stored as a variety of types. Automatic type coercion makes indexing values in Lore more complicated than in traditional database systems. Before discussing the specific "coercible indexes" we have implemented in the Lore system, we present a general framework for indexing coercible values.

3.1 Types and Coercion

Suppose we are given a set of types $\mathcal{T}$, and a partial ordering $\prec$ on the types in $\mathcal{T}$. When $t_1 \prec t_2$, we say that t_1 is a *more specific type* than t_2. This relationship indicates that some (but not necessarily all) values of type t_2 can be coerced into a value of type t_1. In Figure 2(a), for example, we see that integers are more specific than reals and reals are more specific than strings. We call t_1 an *ancestor* of t_2 if $t_1 \prec t_2$ or $t_1 = t_2$. Note that $\prec$ is transitive. The *least upper bound* of types t_1 and t_2, denoted by $\text{lub}(t_1, t_2)$, is the least common ancestor of t_1 and t_2 if one exists.

Each type has associated with it a (usually infinite) set of *values* of that type. We denote the set of values of type t by $\text{DOM}(t)$. Also associated with each pair of types t_i, t_j such that $t_i \prec t_j$ is a *coercion function* f_{ji}, which is a partial function from $\text{DOM}(t_j)$ to $\text{DOM}(t_i)$. We assume coercion functions are transitive, i.e., for a value v, $f_{ij}(f_{jk}(v))$ is defined iff $f_{ik}(v)$ is defined, and when both are defined the coerced values are equal. For convenience, we let f_{ii} denote the identity function for all types t_i. In the example of Figure 2(a) with types `integer`, `real`, and `string`, there are two points to note:

1. The coercion functions are true partial functions; for example, not all strings can be coerced into reals and not all reals can be coerced into integers.

2. The coercion functions are generally many-to-one and therefore not invertible; for example, the strings `"5"` and `"05"` are coerced into the same real number and same integer.

Figure 2(b) shows an example of a nonlinear partial order: certain Postscript and certain MS Word documents can be coerced into plain-text ASCII documents.

3.2 Flexible Comparisons

For developing the theoretical foundation, we say that an OEM database D is a collection of objects of different types. An object is represented by a triple $\langle\, o, t, v\,\rangle$ where o is the OID, $t \in \mathcal{T}$ is a type, and $v \in \text{DOM}(t)$ is a value of type t. The basic *object lookup* problem is to find all atomic objects with a given value v in the database. That is, we wish to compute the set $\{o \mid \langle\, o, t, x\,\rangle \in \mathcal{D} \text{ and } x = v\}$. Obviously, object lookup can be a useful operation when executing a query, since it can directly locate those objects satisfying an equality condition in the query predicate. (We extend to general comparison operators below.)

Often in semistructured data, the same or similar concepts are represented using different types. For example, the number 5 could be represented in some places as the integer 5 and in other places as the string "5". When we perform object lookup for the value 5, we would like to match not only an object with the integer value 5, but also an object with the string value "05". We formalize the idea by defining the notion of *flexible equality*. Suppose $v_1 \in \text{DOM}(t_1)$ and $v_2 \in \text{DOM}(t_2)$. Intuitively, we must "promote" v_1 and v_2 as small a distance up the partial order as possible so that we can compare them. It is important to limit the distance that a value is promoted since the conversion functions are partial functions. We are guaranteed that if $t_3 \prec t_2 \prec t_1$ and $f_{13}(v)$ is defined then $f_{12}(v)$ is defined, but not vice versa since the type t_3 is more specific than t_2.

Formally, we say that v_1 is equal to v_2 under flexible equality, written $v_1 =_f v_2$, if one of the following holds:

1. $t_1 = t_2$ and $v_1 = v_2$.

2. $t_3 = \text{lub}(t_1, t_2)$ exists, $f_{13}(v_1)$ and $f_{23}(v_2)$ are both defined, and $f_{13}(v_1) = f_{23}(v_2)$.

In the partial order shown in Figure 2(a), we have $5 =_f$ "05", because the least upper bound of `integer` and `string` is `integer`, and the coercion of "05" into type `integer` results in the value 5. However, it is not the case that "5" $=_f$ "05" because the least upper bound of `string` and `string` is `string` and "5" $\neq$ "05".

We now generalize flexible equality to flexible comparisons. Suppose $\odot$ is a binary comparison function that is defined over all types. That is, given two values u and v of the same type, $u \odot v$ is either true or false. For example, $\odot$ could be one of the standard arithmetic comparison operators $=, <, >, \leq$, or $\geq$. We define the flexible comparison operator $\odot_f$ corresponding to $\odot$ as follows:

1. $v_1 \odot_f v_2$ if v_1 and v_2 are of the same type and $v_1 \odot v_2$.

2. $v_1 \odot_f v_2$ if v_1 is of type t_1, v_2 is of type t_2, $t_3 = \text{lub}(t_1, t_2)$ exists, $f_{13}(v_1)$ and $f_{23}(v_2)$ are both defined, and $f_{13}(v_1) \odot f_{23}(v_2)$.

Our definition of the flexible comparison operator corresponding to $\odot$ preserves commutativity of $\odot$ but not transitivity. That is, if $\odot$ is commutative, $\odot_f$ is also commutative, but if $\odot$ is transitive, then $\odot_f$ is not in general transitive. For example, "5" $=_f$ 5, 5 $=_f$ "05", but "5" $\neq_f$ "05". Not preserving transitivity does eliminate certain optimization opportunities in query processing, however we feel that this notion of flexible comparisons is appropriate for semistructured data.

	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 integer, real, and string types

3.3 Flexible Indexes

For developing the theoretical foundation, we say that an *index* I for a type t contains a set of objects $\langle O, t, v \rangle$. It efficiently supports the *index lookup* operation for one or more operators $\odot$:

$$L(I, \odot, v) = \{o \mid \langle\, o, t, x \,\rangle \in O \text{ and } x \odot v\}$$

Our goal is thus to construct a set of indexes for a given database such that there is an algorithm to do a "flexible" object lookup of the form $x \odot_f v$ based on index lookups.

We first describe the set of indexes to create, and then in the next subsection the actual lookup algorithm. Let O_i denote the set of objects of type t_i in the database $\mathcal{D}$. For each type t_j that is an ancestor of t_i, we create an index J_{ij}. Let:

$$O_{ij} = \{\langle\, o, t_j, f_{ij}(v) \,\rangle \mid \langle\, o, t_i, v \,\rangle \in O_i \text{ and } f_{ij}(v) \text{ is defined}\}$$

Then J_{ij} is the index for type t_j containing the objects in O_{ij}. In other words, J_{ij} indexes all values of type t_i that can be coerced into type t_j. The coerced values are stored in the index. In the special case where $i = j$ all values of type t_i are indexed.

Example 3.1 For the partial order given in Figure 2(a), we have the following 6 indexes, where we denote the type `string` by s, `real` by r, and `integer` by i:

1. J_{ss} indexes all string values.
2. J_{sr} indexes all strings that can be coerced into reals (as reals).
3. J_{si} indexes all strings that can be coerced into integers (as integers).
4. J_{rr} indexes all reals.
5. J_{ri} indexes all reals that can be coerced into integers (as integers).
6. J_{ii} indexes all integers.

It is easy to compute the number of indexes in a partial order such as the one in Figure 2(a). There is one index associated with the top type, two indexes associated with the second type, and so on. In general, if there are n types, the number of indexes is $\frac{n(n+1)}{2}$. The number of indexes is less than this number for a nonlinear partial order with n types. For example, there are 5 indexes for Figure 2(b).

It is possible to reduce the number of indexes if certain properties hold. In particular, suppose types t_1 and t_2 are such that:

1. $t_1 \prec t_2$; and

2. there exists a coercion function f_{12} from $\text{DOM}(t_1)$ to $\text{DOM}(t_2)$ such that for every pair of values $u, v \in \text{DOM}(t_1)$, $f_{12}(u)$ and $f_{12}(v)$ exist and $u \odot v$ iff $f_{12}(u) \odot f_{12}(v)$.

In this case we can "fold" the indexes for t_1 into the indexes for t_2. For example, the types `integer` (t_1) and `real` (t_2) satisfy the above conditions for the operators $=, <, >, \leq, \geq$, so we can fold all integer indexes into real indexes. This reduces the total number of indexes in Example 3.1 from six to three, keeping items 1, 2 and 4 (where 4 now contains all integers coerced to reals as well as all reals).

Using this optimization, the full set of coercion rules is summarized in Table 1, and this is the approach we take to value indexing in Lore.

3.4 The Index Lookup Algorithm

Suppose we wish to look up value v_i of type t_i under operator $\odot_f$. The algorithm is displayed in Figure 3. Recall that $L(J, \odot, v)$ is the lookup operation in index J on value v with operator $\odot$. The set S returned contains the set of objects that "match" the value v_i under the flexible comparison operator $\odot_f$. An interesting point is that for a partial order of n types, at most n index lookups are needed regardless of the type t_i. An index lookup need not occur when two types do not have a least upper bound.

Procedure *IndexLookup*(Value v_i, Type t_i, Operator $\odot$)
$S := \emptyset$
foreach Type t_j
  $t_k = \text{lub}(t_i, t_j)$.
  if $t_k \neq \emptyset$ **and** $f_{ik}(v_i)$ is defined **then**
    $S_j := L(J_{jk}, \odot, f_{ik}(v_i))$.
    $S := S \cup S_j$.
return S

Figure 3: Index lookup algorithm

Example 3.2 Suppose we wish to match the `string` value "05" given the partial order shown in Figure 2(a) without the optimization discussed in Section 3.3. Procedure *IndexLookup* would generate the following index lookups: (i) Lookup "05" in J_{ss}; (ii) Lookup 5.0 in J_{rr}; (iii) Lookup 5 in J_{ii}.

Example 3.3 Suppose we wish to match the `real` number 5.0 given the partial order shown in Figure 2(a) without the optimization discussed in Section 3.3. Procedure *IndexLookup* would generate the following index lookups: (i) Lookup 5.0 in J_{sr}; (ii) Lookup 5.0 in J_{rr}; (iii) Lookup 5 in J_{ii}.

Theorem 3.1 At the end of executing Procedure *IndexLookup*, the set S contains all and only the objects that match v_i under $\odot_f$. Moreover, each union in line (6) of `IndexLookup` is a disjoint union; that is, $S_j \cap S_k = \emptyset$ for $j \neq k$.

Proof: The result of each lookup is:

$$\begin{aligned} S_j &= L(J_{jk}, \odot, f_{ik}(v_i)) \\ &= \{o \mid \langle\, o, t_j, v_j \,\rangle \in O_{jk} \text{ and } v_j \odot_f f_{ik}(v_i)\} \end{aligned}$$

The fact that the set S contains all and only the objects that match v_i under $\odot_f$ follows directly from the definition of S_j. To see why each union is a disjoint union, suppose $S_j \cap S_k \neq \emptyset$ for some $j \neq k$. Then there must exist some o that is returned by indexes J_{xy} and J_{wz}. Recall that J_{st} is the index for objects of type s that have been coerced into type t. Since an object only has a single type then $x = w$. Now suppose $y \neq z$ (if not then $S_j = S_k$ and we are done). But this is impossible since line (5) only uses x a single time, and $y \neq z$ implies that there are two least upper bounds for $\text{lub}(t_i, t_j)$, a contradiction. □

The fact that we are computing disjoint unions is significant, because it shows that we are doing as little work as possible: we never obtain the same matching object from two different index lookups. Moreover, it also reduces the complexity of the algorithm because disjoint unions are much easier to compute than general unions.

4 Indexes In Lore

We now describe the types of indexes that can be built over a Lore database. We will then describe the query plan operators that use the index structures, followed by a description of the query plans themselves. To speed up query processing in a Lore database, we can build four different types of index structures. The first two identify objects that have specific values; the next two are used to efficiently traverse the database graph.

1. A value index, or *Vindex*, locates atomic objects with certain values. *Vindexes* are based on the coercible indexing scheme introduced in Section 3. *Vindexes* can be built selectively over those objects with certain incoming labels.

2. A text index, or *Tindex*, locates string atomic values containing specific words or groups of words. The text index is a simplified version of information-retrieval style inverted indexes [Sal89]. Like *Vindexes*, *Tindexes* can be built selectively over those objects with certain incoming labels.

3. Since our current implementation of OEM does not support parent pointers, a link index, or *Lindex*, locates parents of a specific object.

4. A path index, or *Pindex*, provides fast access to all objects reachable via a given labeled path.

As in all database systems, the choice of which indexes to build and maintain is made by the database administrator based on expected queries and updates. We now describe each of these types of indexes in more detail.

4.1 Value Index (Vindex)

A *Vindex* in Lore is built over all atomic objects of base type `integer`, `real`, or `string` that have an incoming edge with a given label l.[2] This *Vindex* allows the query engine to quickly locate all objects reachable by an l edge and matching a comparison predicate. While we could have chosen to support a single label-independent *Vindex*, a specific desired incoming label usually is known at query processing time (see Section 5.2), so it is useful to partition the *Vindex* by label. This approach also allows the administrator to build *Vindexes* selectively for frequently used labels. Our *Vindex* query operator does allow the label to be omitted, in which case all *Vindexes* are searched if they exist for all labels. *Vindexes* use the coercible indexing scheme described in Section 3, with the optimization outlined in Section 3.3. Thus, if the administrator requests a *Vindex* for label l, we actually create three indexes: one for reals and integers coerced to reals, one for strings, and one for strings coerced to reals. Each index is implemented as a B+-tree to support inequality as well as equality lookups. The lookup procedure for *Vindexes* is similar to the *IndexLookup* procedure given in Figure 3, except we extend it slightly to accept a label.

Example 4.1 Suppose we create a *Vindex* for incoming label *Price* over the database in Figure 1. If we perform a lookup for values > 15.00 with incoming edge *Price*, the result is $\{\&11, \&15\}$.

4.2 Text Index (Tindex)

A *Vindex* is useful for finding values that satisfy basic comparisons such as $=, <$, etc. However, for string values an information-retrieval style keyword search can be very useful, especially for strings containing a significant amount of text. In these situations, the *Vindex* is not powerful enough and a different indexing structure, the *Tindex*, is used.

Text indexes in Lore are implemented using inverted lists, which map a given word w and label l to a list of atomic values with incoming edge l that contain word w. Like the *Vindex*, *Tindexes* are created by the administrator for a given label, for the reasons outlined earlier, but the label can always be omitted for a full search. A *Tindex* lookup returns a list of *postings*, where each posting is of the form $\langle\, o, n\,\rangle$. A posting indicates that w appears in object o as the nth word in the value, and o has an incoming edge labeled l. The inverted lists are stored in hash tables on disk keyed on w.

Example 4.2 Consider a *Tindex* lookup for all objects with an atomic string value containing the word "Ford" and an incoming edge *Name*, performed over the database in Figure 1. The result is $\{\langle\,\&17, 2\,\rangle, \langle\,\&21, 2\,\rangle\}$.

[2] Lore currently does not support value indexing of "novel" types such as `gif`, `audio`, etc.

As will be seen in Section 5.1, a higher-level query plan operator above the *Tindex*, and a corresponding operator in the query language, support additional text search features such as `AND`, `OR`, `NEAR`, etc.

4.3 Link Index (Lindex)

Since we do not support inverse pointers in OEM graphs, the *Lindex* provides a mechanism for retrieving the parents of an object via a given label. A *Lindex* lookup takes a "child" object c and a label l, and returns all parents p such that there is an l-labeled edge from p to c. The *Lindex* also supports lookups with no label, in which case all parents and their labels are returned. In Lore, the *Lindex* is implemented using extendible hashing [FNPS79] since for the *Lindex* we are always doing equality lookups. Currently we do not support selective creation of *Lindexes*, i.e., one *Lindex* is created for the entire database graph.

Example 4.3 Suppose we had located all atomic objects containing the word "Ford" via the *Tindex* lookup in Example 4.2, and we now wanted to traverse up to parent subobjects connected via the label *Name*. The *Lindex* lookup for object &17 returns parent object &6, and the lookup for object &21 returns object &13.

As will be seen in Section 5.1, a higher-level query plan operator above the *Lindex* supports finding ancestors objects that are reached by following a component of a general path expression that may include regular expression operations.

4.4 Path Index (Pindex)

Finding all objects reachable by a given labeled path through the database is an important part of query processing. A *Pindex* lookup for a path p returns the set of objects O reachable via p. Currently in Lore we only index those paths that begin at named objects and contain no regular expressions. By doing so, we can easily store the set of reachable objects for each path p in Lore's *DataGuide* [GW97]. The DataGuide is a dynamic structural summary of all possible paths within the database at any given point in time. In addition to providing a kind of dynamic "schema" for the user to browse, the DataGuide also stores OIDs and statistics for those objects reachable via each path beginning from a name. The algorithm to compute as well as incrementally maintain the DataGuide *Pindex* appears in [GW97].

Example 4.4 Suppose our query is simply "**select** DB.Movie.Title" applied over the database in Figure 1. Instead of exploring the graph, we can use the *Pindex* to directly locate all objects reachable via `DB.Movie.Title`. The *Pindex* lookup operation returns $\{\&5, \&9, \&14\}$.

While the *Pindex* may appear very attractive, we cannot use it for all queries and path expressions: in addition to the limitations above, in some queries we also must obtain the objects along a given path in addition to those at the end of the path. See [MW97] for details.

DB	OA[1].Movie	OA[2].Title	OA[2].Actor	OA[4].Name	Exists OA[5]?
OA[1]	OA[2]	OA[3]	OA[4]	OA[5]	OA[6]

Figure 4: Object Assignment for Example 2.2

5 Query Plans Using Indexes

A query plan in the Lore system is a tree of query operators that describes the specific sequence of steps used to answer a query. 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. 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. Intuitively, each slot within an OA holds the OID of a node on a data path currently being considered by the query engine. Figure 4 contains an OA that will be described in more detail later. At a given point during query processing, not all slots of the current OA necessarily contain a valid OID; 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.

5.1 Query Operators

We briefly explain some of the key query operators that can appear in our query plans, especially those related to indexing. Each operator takes a number of arguments. If an operator produces a result then the last argument is the OA slot that contains the result. Some operators do not produce a result but affect the flow of control when executing the plan.

Each of the indexing structures introduced in Section 4, the *Vindex*, *Tindex*, *Lindex*, and *Pindex*, have a corresponding query operator that supports the appropriate lookup operation during query execution. The `Vindex` and `Pindex` operators are simple "wrappers" around the corresponding index structures. We discuss the `Lindex` and `Tindex` operators in more detail since they add additional capabilities beyond what the corresponding indexes support.

Like the *Lindex* data structure, the `Lindex` query plan operator can be used to find the l-labeled parents of an object o. In addition, the `Lindex` operator can also find the ancestor objects of o that are reached by matching some path described by a component of a general path expression. For example, the `Lindex` can find all ancestors of o that are reached by following the path `(.A|.B)*`, which matches any path having zero or more A or B labeled edges. The `Lindex` operator keeps a run-time stack of objects visited in order to match a sequence of zero or more edges. (Since we do not yet support full regular expressions, a complete finite-state automaton is not required.)

The `Tindex` operator uses the *Tindex* data structure, but it adds considerable power beyond simple single-word searches. Specifically, the `Tindex` operator takes three arguments, a text *pattern* to match, a label, and a destination OA slot that will contain the OIDs of all matching string atomic objects with the given incoming label. The possible patterns are defined recursively:

- *Word* or *Phrase* match. A string matches a word or phrase iff it contains the word or phrase. The *Tindex* operator supports both case-sensitive and case-insensitive match.
- *NEAR* match. A string matches *phrase1* NEAR *phrase2* iff it matches both *phrase1* and *phrase2*, and the two matching phrases in the string are within 10 words of each other.
- *AND* operator. A string matches *phrase1* AND *phrase2* iff it matches both *phrase1* and *phrase2*.
- *OR* operator. A string matches *phrase1* OR *phrase2* iff it matches either *phrase1* or *phrase2*.
- *ANDNOT* operator. A string matches *phrase1* ANDNOT *phrase2* iff it matches *phrase1* but not *phrase2*.

While this functionality is a simple subset of that offered by, e.g., Web search engines, we have found it very useful in practice to speed up a large class of common queries over string or text values.

In addition to the four index operators, there are many other operators to perform the standard tasks necessary in query processing. (For complete details see [MW97].) For example, the `Scan` operator returns all OIDs that are subobjects of a given object following a specified path expression. It accepts an object o and a component of a path p and returns the set of objects that are reachable starting from o and matching p. Like the `Lindex` operator, the `Scan` operator may match a path containing regular expression operators by using a simple run-time stack.

The `Join`, `Project`, and `Select` operators are nearly identical to their corresponding relational operators. Like a relational nested-loop join, the `Join` operator 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` operator is used to limit which objects should be returned by specifying a set of OA slots, while the `Select` operator applies a predicate to the object identified by the OID in the specified OA slot.

The `Aggregation` operator implements standard aggregation operations as well as existential quantification. At a high level, the aggregation operator 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. For the aggregation "operation" *Exists* the operator adds *true* if the existential quantification is satisfied and *false* otherwise. Filtering of OAs whose quantification is *true* occurs in a `Select` operator which must appear 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.

5.2 Query Plans

We present Lore query plans through several examples. There are numerous possible plans for each query, and Lore includes a traditional cost-based query optimizer for plan enumeration and selection

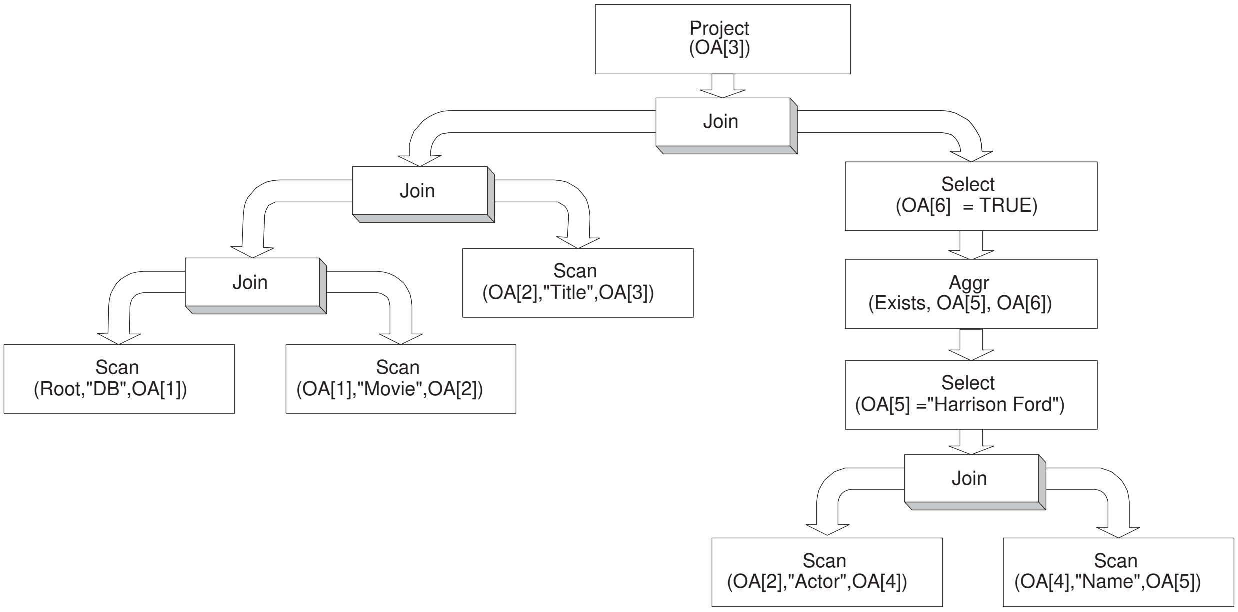

Figure 5: Top-down plan for query in Example 2.2

as detailed in [MW97]. Here we focus on describing how query plans exploit indexes. Recall the query introduced in Example 2.2:

> **select** T
> **from** DB.Movie M, M.Title T
> **where** exists A in M.Actor : exists N in A.Name : N = "Harrison Ford"

A possible OA structure for this query is shown in Figure 4. Notice that OA[6] is used for the result of the aggregation operation. A specific OA structure is created for each query plan. The OA in Figure 4 works with the "top-down" query plan explained next.

5.2.1 Top-Down Query Plan

A top-down strategy (as introduced in Section 2.3) for this query does not exploit indexes. It attempts to match the path in the `from` clause first by providing bindings for OA[1], then OA[2], then OA[3]. The `where` clause is handled in a similar fashion, and finally the result is returned. This execution strategy is reflected in the query plan shown in Figure 5. The left subtree of the top `Join` node handles the `from` clause by first scanning for the named object *DB*. From the resulting object of that scan the second `Scan` operator looks for a *Movie* subobject and places the result in OA[2]. The third `Scan` retrieves the movie's *Title* subobject. The right subtree first looks for an *Actor* subobject of the current movie, then the bottom-right `Scan` and the `Select` operators find all *Name* subobjects of the object in OA[4] with value "Harrison Ford". To satisfy the existential condition, the query plan uses the `Aggregation/Select` sequence, as described in Section 5.1. Note

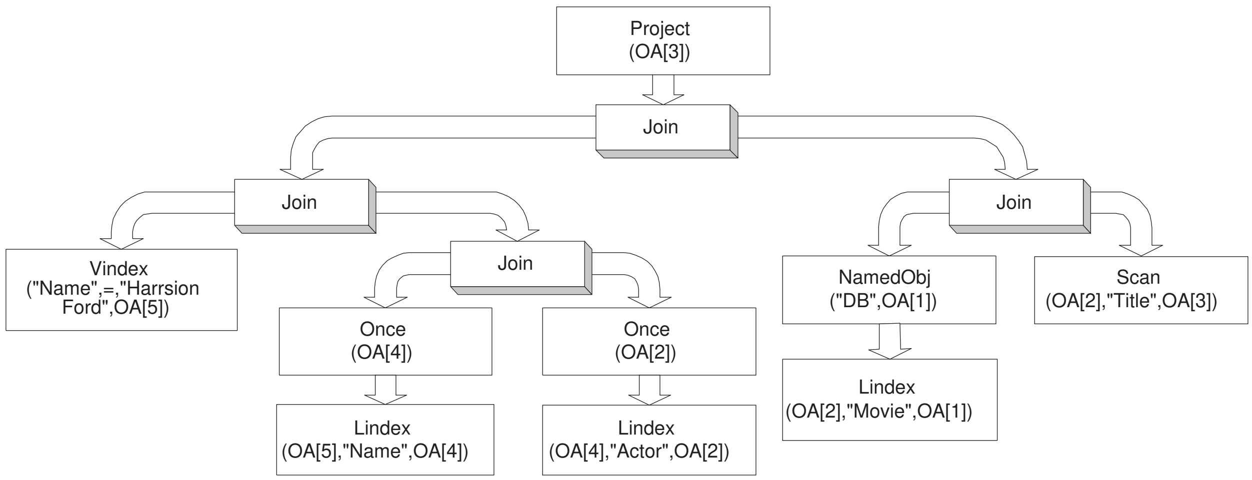

Figure 6: Bottom-up plan for query in Example 2.2

that in this query plan it is not necessary to test the existential condition for *Actor*, since if no actor subobject exists then we would never be able to satisfy the existential condition for *Name*.

This query plan performs poorly if there are many movies in the database that "Harrison Ford" did not appear in, since in that case it would traverse many paths unnecessarily.

5.2.2 Bottom-Up Query Plan

We now consider a very different query plan for the same query, which uses some of Lore's indexing structures through their corresponding index operators. Refer to Figure 6. In this plan the left subtree of the top-most `Join` operator first satisfies the `where` clause. The `Vindex` operator will find all atomic objects with an incoming edge *Name* whose value is "Harrison Ford". The `Lindex` operators then attempt to match backwards the edges *Name* and then *Actor* starting from the objects identified via the `Vindex` operator. The `Once` operator is the bottom-up equivalent of existential quantification: it ensures that each object is passed up to the parent at most once. After the `where` clause has been processed, we match a *Movie* edge via the third *Lindex* operator. It is then necessary to confirm that the parent of the movie object is the named object *DB*, accomplished via the `NamedObject` operator. Since the query requests all *Title* subobjects of a movie, we use the `Scan` operator to obtain the title.

This query plan performs poorly if there are many objects with value "Harrison Ford" and incoming edge *Name*, but few that are reachable via the path `DB.Movie.Actor.Name`.

5.2.3 Hybrid Query Plan

In certain cases, it is advantageous to combine the two techniques described so far, bottom-up and top-down, into a single *hybrid* plan. For instance, it may be beneficial to first locate all objects that satisfy the `where` clause via `Vindex` and `Lindex` operators and then, instead of continuing a bottom-

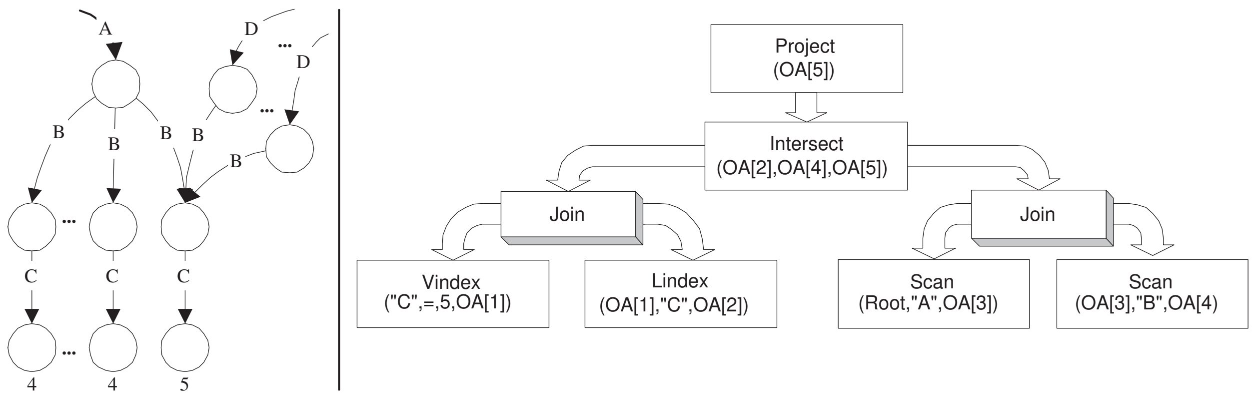

Figure 7: A database where the hybrid execution strategy is preferred along with a hybrid query plan

up traversal, identify all objects that satisfy the `from` clause in a top-down fashion. Consider the following simple query:

select X
from A.B X
where exists Y in X.C : Y = 5

Now consider the database and query plan in Figure 7. A top-down strategy would visit all the leaf objects, but only a single one satisfies the predicate. A bottom-up strategy would identify the single object satisfying the predicate, but would then unnecessarily visit all of the nodes in the upper right portion of the database. In this case, a hybrid plan where we use bottom-up execution to find the objects satisfying the `where` clause, then top-down execution to obtain the set of all `A.B` objects, then finally intersect the sets, would be a good query execution strategy. The hybrid query plan is shown in Figure 7. Notice the `Intersect` operation splits the execution of the `where` clause (appearing on the left) and the `from` clause (appearing on the right).

5.2.4 Full Text and Path Indexing

To illustrate the use of text and path indexes, suppose we would like to find all movies such that a *Title* or *Description* subobject contains the word "black" near the word "sheep". This query can be written as follows, where `hasword` is the Lorel query operator that exposes the functionality of our text indexing system:

select DB.Movie
where DB.Movie(.Title—.Description) hasword "black NEAR sheep"

In Figure 8, we present a hybrid query plan that uses the *Pindex* and *Tindex*. The left subplan of the `Intersect` operator identifies (using the *Lindex*) all parents of objects that are connected via an edge *Title* or *Description* to a child containing the word "black" near the word "sheep" (located via the *Tindex*). The right child of the `Intersect` operator identifies all objects reachable via the

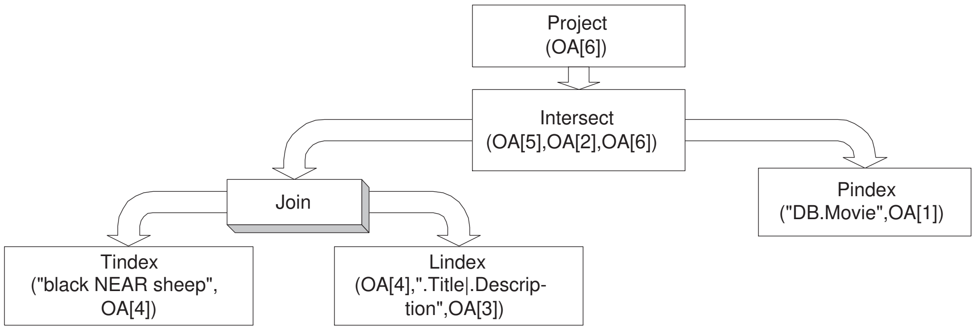

Figure 8: A query plan that uses both *Tindex* and *Pindex* operators

path `DB.Movie`, found using the *Pindex* operator. The intersection of the results generated by each subplan produces the answer to the query.

6 Conclusion

This paper presented the management of indexes in Lore, a DBMS for semistructured data. We provided a general framework for indexing values in the presence of automatic type coercion. Lore's *Vindex*, *Tindex*, *Lindex*, and *Pindex* structures were introduced. We then described various query plans in Lore and how they make use of indexes. All indexing structures, index maintenance, query operators, and query plans described in this paper are fully implemented within the Lore system. Preliminary performance results indicate that at least an order of magnitude improvement is observed on a wide class of common databases and queries when indexes are used for query processing.[3]

References

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

[BG92] E. Bertino and C. Guglielmina. Optimization of object-oriented queries using path indices. In *Proceedings of the Second International Workshop on Research Issues in Data Engineering*, pages 140–149, February 1992.

[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.

[3] While time constraints prevented us from including a full performance study in this paper, we currently are conducting such a study and results could be added to the final published report.

[CCY94] S. S. Chawathe, M. S. Chen, and P. S. Yu. On index selection schemes for nested object hierarchies. In *Proceedings of the Twentieth International Conference on Very Large Data Bases*, pages 331–341, Santiago, Chile, September 1994.

[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, 1998. To appear in *Proceedings of the Fourteenth International Conference on Data Engineering*, Orlando, Florida, February 1998.

[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.

[KKD89] W. Kim, K.-C. Kim, and A. Dale. Indexing techniques for object-oriented databases. In W. Kim and F.H. Lochovsky, editors, *Object-Oriented Concepts, Databases and Applications*, chapter 15. Addison-Wesley, 1989.

[KM92] A. Kemper and G. Moerkotte. Access support relations: An indexing method for object bases. *Information Systems*, 17(2):117–145, 1992.

[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.

[MW97] J. McHugh and J. Widom. Query optimization in semistructured data. Technical report, Stanford University Database Group, 1997. Available at `http://www-db.stanford.edu/pub/papers/qo.ps`.

[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 (DOOD)*, pages 319–344, Singapore, December 1995.

[RK95] S. Ramaswamy and P. C. Kanellakis. OODB indexing by class-division. *SIGMOD Record*, 24(2):139–150, June 1995.

[Sal89] Gerard Salton. *Automatic Text Processing: The transformation, analysis, and retrieval of information by computer.* Addison-Wesley, 1989.

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

[SS94] B. Sreenath and S. Seshadri. The hcC-tree: An efficient index structure for object oriented databases. In *Proceedings of the Twentieth International Conference on Very Large Data Bases*, pages 203–213, Santiago, Chile, September 1994.

[XH94] Z. Xie and J. Han. Join Index Hierarchies for Supporting Efficient Navigations in Object-Oriented Databases. In *Proceedings of the Twentieth International Conference on Very Large Databases*, pages 522–533, Santiago, Chile, September 1994.