DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases*

Roy Goldman
Stanford University
royg@cs.stanford.edu

Jennifer Widom
Stanford University
widom@cs.stanford.edu

Abstract

In *semistructured* databases there is no schema fixed in advance. To provide the benefits of a schema in such environments, we introduce *DataGuides*: concise and accurate structural summaries of semistructured databases. DataGuides serve as dynamic schemas, generated from the database; they are useful for browsing database structure, formulating queries, storing information such as statistics and sample values, and enabling query optimization. This paper presents the theoretical foundations of DataGuides along with algorithms for their creation and incremental maintenance. We provide performance results based on our implementation of DataGuides in the *Lore* DBMS for semistructured data. We also describe the use of DataGuides in Lore, both in the user interface to enable structure browsing and query formulation, and as a means of guiding the query processor and optimizing query execution.

1. Introduction

Traditional relational and object-oriented database systems force all data to adhere to an explicitly specified schema. Yet a typical site on the World-Wide Web demonstrates that much of the information available on-line is *semistructured*. Although the data may exhibit some structure, it is too varied, irregular, or mutable to easily map to a fixed schema. Recent research has focused on data models, query languages, and systems that do not require a schema to accompany each database [AQM+96, BDHS96, BDS95, KS95, MAG+97].

Beyond its use to define the structure of the data, a schema serves two important purposes:

- A schema, in the form of either tables and their attributes or class hierarchies, enables users to understand the structure of the database and form meaningful queries over it.
- A query processor relies on the schema to devise efficient plans for computing query results.

Without a schema, both of these tasks become significantly harder. Although it may be possible to manually browse a small database, in general forming a meaningful query is difficult without a schema or some kind of structural summary of the underlying database. Further, a lack of information about the structure of a database can cause a query processor to resort to exhaustive searches. To address these challenges in "schema-free" environments, we introduce *DataGuides*, dynamically generated and maintained structural summaries of semistructured databases. This paper makes several contributions:

- We give a formal definition of DataGuides as concise, accurate, and convenient summaries of semistructured databases. Further, we motivate and define *strong* DataGuides, well-suited for implementation within a DBMS.
- We provide simple algorithms to build strong DataGuides and keep them consistent when the underlying database changes.
- We show how to store sample values and other statistical information in a DataGuide.
- We demonstrate how DataGuides have been successfully integrated into *Lore* [MAG+97] (for *Lightweight Object Repository*), a DBMS for semistructured data under development at Stanford University. DataGuides are vital to Lore's user interface: users depend on the DataGuide to learn

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

about the structure of a database so they can formulate meaningful queries. In addition, users may specify and submit queries directly from the DataGuide.

- Finally, we explain how a query processor can use a strong DataGuide to significantly optimize query execution.

Our work is cast in the context of the Lore system. All data in Lore follows a simple, graph-based data model called *OEM,* for *Object Exchange Model* [PGW95]. Thus, our work can be applied easily to any graph-based data model. A Lore database is queried using *Lorel* [AQM+96], an OQL-based language designed for easy and effective querying over semistructured data. Though initially designed as a lightweight, read-only system, we are steadily adding traditional DBMS features to Lore, such as update support, concurrency control and transaction management.

Within Lore, DataGuides serve much the same role as traditional metadata. For example, DataGuides are stored directly in Lore as OEM objects. As with metadata in relational or object-oriented systems, user interfaces or client applications may access and query the DataGuide through Lore's standard interfaces [MAG+97]. And in the same way that a traditional query processor consults metadata, the DataGuide is available to guide Lore's query processor. Of course, DataGuides also differ significantly from metadata, since they are dynamically generated: DataGuides conform to the data, rather than forcing data to conform to the DataGuides.

1.1 Related Work

DataGuides extend initial work presented in [NUWC97], which gives a theoretical foundation to the concept of dynamically generated structural summaries of graph-structured databases, called *Representative Objects* (ROs). Their foundational work defines these summaries in a functional style, with less emphasis on implementation; experimental performance and incremental maintenance are not addressed. While ROs are shown to be useful to guide query formulation and optimization, DataGuides significantly extend these contributions.

Other related theoretical research is presented in [BDFS97], which discusses schemas for graph-structured databases. A formal definition of a *graph schema* is given, along with an algorithm to determine whether a database conforms to a specific schema. Schema ordering, subsumption, and equivalence are also discussed. The work in [BDFS97] is presented with a more traditional view of a schema than we take. Optimization and browsing functionality depend on having a database (or at least large fragments of the database) conform to an explicitly specified schema. In contrast, our work focuses directly on the case where it is inconvenient or implausible to specify and maintain a schema: DataGuide summaries are dynamically generated and maintained to always represent the current state of the database. A DataGuide never includes information that does not exist in the database, and by definition any database always "conforms" to its DataGuide. A graph schema, on the other hand, could be a superset of any database that conforms to it, and complications are incurred if a database changes and no longer conforms to that schema.

As with many research and commercial user interfaces that use a schema (or structural summary) to guide browsing and query formulation, our work has been influenced by the seminal work on *Query By Example* [Zlo77]. In addition to early research efforts such as Timber [SK82], many commercial relational front-ends such as Access and Paradox have sophisticated interfaces for visually specifying queries. Several visual database browsers have also been developed for richer, object-oriented data models, including KIVIEW [MDT88] and OdeView [AGS90]. PESTO [CHMW96] is a visual tool for exploring object databases that integrates browsing and querying into a single interface. The DataGuide is unique as a graphical browsing and query tool, since it presents a template dynamically generated directly from a database without regard to any fixed schema or class hierarchy.

For query optimization, we show how the DataGuide can be used as a *path index*. Substantial research on object-oriented query optimization has focused on the design and use of path indexes, e.g., [BK89, CCY94, KM92]. In general, previous work has been based on the class hierarchies of the object-oriented model. The

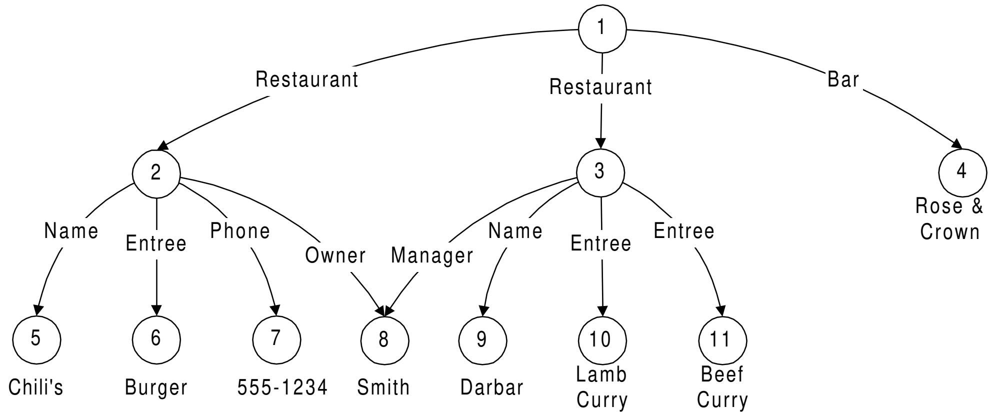

Figure 1. A sample OEM database

issues of how to create, maintain, and use a path index in a semistructured data model such as OEM have not to the best of our knowledge been addressed.

1.2 Paper Outline

Section 2 first reviews the data model and query language with which we are working. It then provides the motivation and definition for DataGuides, along with a simple algorithm for creating them. In Section 3 we present experimental results showing the time and space required to build and store typical DataGuides. Section 4 presents an incremental algorithm for DataGuide maintenance in response to database modifications. Section 5 describes how DataGuides are used in practice to browse structure and guide query formulation through a graphical interface to the Lore system. In Section 6 we see how a strong DataGuide can improve query processing in Lore. We conclude the paper and discuss future research in Section 7.

2. Foundations

In this section we describe our basic data model and query language. We then motivate and define DataGuides and their properties, and we provide an algorithm for building them.

2.1 Object Exchange Model

Our research is based on the *Object Exchange Model* (*OEM*), a simple and flexible data model that originates from the *Tsimmis* project at Stanford University [PGW95]. OEM itself is not particularly original, and the work presented here adapts easily to any graph-structured data model. In OEM, each object contains an object identifier (oid) and a value. A value may be atomic or complex. Atomic values may be integers, reals, strings, images, programs, or any other data considered indivisible. A complex OEM value is a collection of 0 or more OEM subobjects, each linked to the parent via a descriptive textual label. Note that a single OEM object may have multiple parent objects and that cycles are allowed. For more details on OEM and its motivation see [AQM+96, PGW95].

Figure 1 presents a very small sample OEM database, representing a portion of an imaginary eating guide database. Each object has an integer oid. Our database contains one complex root object with three subobjects, two Restaurants and one Bar. Each Restaurant is a complex object and the Bar is atomic, containing the string value "Rose & Crown." Each Restaurant has an atomic Name. The Chili's restaurant has atomic data describing its Phone number and one available Entree. We can see that the database structure is irregular, since restaurant Darbar, with two Entrees, doesn't include any phone number information. Finally, we see that OEM databases need not be tree-structured—Smith is the Owner of one restaurant and Manager of the other.

Next, we give several simple definitions useful for describing an OEM database and subsequently for defining DataGuides.

Definition 1. A *label path* of an OEM object o is a sequence of one or more dot-separated labels, $l_1.l_2...l_n$, such that we can traverse a path of n edges ($e_1...e_n$) from o where edge e_i has label l_i. ❑

In Figure 1, Restaurant.Name and Bar are both valid label paths of object 1. In an OEM database, queries are based on label paths. For example, in Figure 1, a valid query might request the values of all Restaurant.Entree objects that satisfy a given condition. Queries are further discussed in Section 2.2.

Definition 2. A *data path* of an OEM object o is a dot-separated alternating sequence of labels and oids of the form $l_1.o_1.l_2.o_2 ...l_n.o_n$ such that we can traverse from o a path of n edges ($e_1...e_n$) through n objects ($x_1...x_n$) where edge e_i has label l_i and object x_i has oid o_i. ❑

In Figure 1, Restaurant.2.Name.5 and Bar.4 are data paths of object 1.

Definition 3. A data path d is an *instance* of a label path l if the sequence of labels in d is equal to l. ❑

Again in Figure 1, Restaurant.2.Name.5 is an instance of Restaurant.Name and Bar.4 is an instance of Bar.

Definition 4. In an OEM object s, a *target set* is a set t of oids such that there exists some label path l of s where $t = \{o \mid l_1.o_1.l_2.o_2...l_n.o$ is a data path instance of $l\}$. That is, a target set t is the set of all objects that can be reached by traversing a given label path l of s. We also say that t is "the target set of l in s," and we write $t = T_s(l)$. We say that l *reaches* any element of t, and likewise each element of t is *reachable* via l. ❑

For example, the target set of Restaurant.Entree in Figure 1 is {6, 10, 11}. Note that two different label paths may share the same target set. {8}, for instance, is the target set of both Restaurant.Owner and Restaurant.Manager.

2.2 Lorel Query Language

Lorel (for *Lore language*) was developed at Stanford to enable queries over semistructured OEM databases. Lorel is based on OQL [Cat93], with modifications and enhancements to support semistructured data; for details see [AQM+96]. As an extremely simple example, in Figure 1 the Lorel query

```
Select Restaurant.Entree
```

returns all entrees served by any restaurant, the set of objects {6, 10, 11}. As another simple example, we may request the names of all restaurants that serve burgers:

```
Select Restaurant.Name
Where  Restaurant.Entree = "Burger"
```

In Figure 1, the answer to the query is the single object 5.

As these brief examples indicate, some knowledge of the structure of the database is important for forming meaningful queries. The Lorel language does provide several facilities, such as "wildcards" in label paths, to enable queries when the database structure isn't entirely known. Still, a summary of the structure of the underlying database is invaluable for guiding the formulation of meaningful queries in Lorel.

2.3 DataGuides

We are now ready to define a DataGuide, intended to be a *concise*, *accurate*, and *convenient* summary of the structure of a database. Hereafter, we refer to a database that we summarize as the *source database*, or simply the *source*. We assume a given source database is identified by its root object. To achieve *conciseness*, we specify that a DataGuide describes every unique label path of a source exactly once, regardless of the number of times it appears in that source. To ensure *accuracy*, we specify that the DataGuide encodes no label path that does not appear in the source. Finally, for *convenience*, we require that a DataGuide itself be an OEM object so we can store and access it using the same techniques available for processing OEM databases. The formal definition follows.

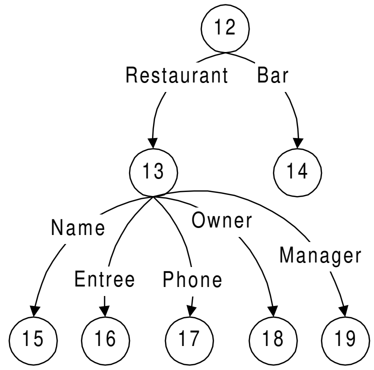

Figure 2. A DataGuide for Figure 1

Definition 5. A *DataGuide* for an OEM source object *s* is an OEM object *d* such that every label path of *s* has exactly one data path instance in *d*, and every label path of *d* is a label path of *s*. ❏

Figure 2 shows a DataGuide for the source OEM database shown in Figure 1. Using a DataGuide, we can check whether a given label path of length *n* exists in the original database by considering at most *n* objects in the DataGuide. For example, in Figure 2 we need only examine the outgoing edges of objects 12 and 13 to verify that the path Restaurant.Owner exists in the database. Similarly, if we traverse the single instance of a label path *l* in the DataGuide and reach some object *o*, then the labels on the outgoing edges of *o* represent all possible labels that could ever follow *l* in the source database. In Figure 2, the five different labeled outgoing edges of object 13 represent all possible labels that ever follow Restaurant in the source. Notice that the DataGuide contains no atomic values. Since a DataGuide is intended to reflect the structure of a database, atomic values are unnecessary. Later we will see how special atomic values, when added to DataGuides, can play an important role in query formulation and optimization. Note that every target set in a DataGuide is a singleton set. Recalling Definition 4, a target set denotes all objects reachable by a given label path. Since any label path in a DataGuide has just one data path instance, the target set contains only one object—the last object in that data path.

A considerable theoretical foundation behind DataGuides can be found in [NUWC97]. That paper proved that creating a DataGuide over a source database is equivalent to conversion of a non-deterministic finite automaton (NFA) to a deterministic finite automaton (DFA), a well-studied problem [HU79]. When the source database is a tree, this conversion takes linear time. However, in the worst case, conversion of a graph-structured database may require time (and space) exponential in the number of objects and edges in the source. Despite these worst-case possibilities, experimental results in Section 3 are encouraging, indicating that for typical OEM databases, the running time is very reasonable and the resulting DataGuides are significantly smaller than their sources. (Unfortunately, no research known to the authors formally identifies those NFAs that do or do not require exponential time or space to be converted to equivalent DFAs.) Work in [NUWC97] focuses on the benefits of relaxing the DataGuide definition to enable a more compact, and sometimes faster-to-create, structural summary called a *k-Representative Object (k-RO).* A k-RO may describe a superset of the label paths that exist in the source, therefore violating the *accuracy* constraint of our DataGuide definition. The k-RO can still be useful to guide query formulation, but DataGuide accuracy is crucial for the query optimization features we discuss in Section 6. Here, we concentrate on (accurate) DataGuides, which in the common case are reasonably small and fast to create.

2.4 Existence of Multiple DataGuides

From automata theory, we know that a single NFA may have many equivalent DFAs [HU79]. Similarly, as shown in Figure 3, one OEM source database may have multiple DataGuides. Figures 3(b) and (c) are both DataGuides of the source in Figure 3(a). Each label path in the source appears exactly once in each DataGuide,

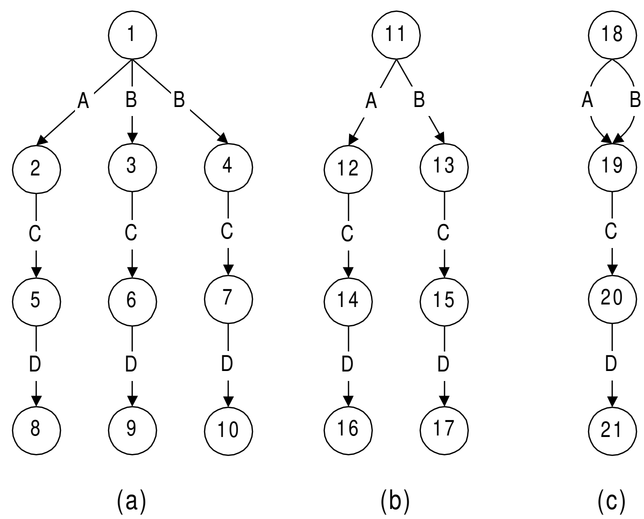

Figure 3. A source and two DataGuides

and neither DataGuide introduces any label paths that do not exist in the source. Figure 3(c) is in fact *minimal*: the smallest possible DataGuide. (Well-known state minimization algorithms can be used to convert any DataGuide into a minimal one [Hop71].) Given the existence of multiple DataGuides for a source, it is important to decide what kind of DataGuide should be built and maintained in a semistructured database system. Intuitively, a minimal DataGuide might seem desirable, furthering our goal of having as concise a summary as possible; [NUWC97] also suggests building a minimal DataGuide. Yet, as we now explain, a minimal DataGuide is not always best.

First, incremental maintenance of a minimal DataGuide can be very difficult. In Figure 3(a), suppose we add a new child object to 10, via the label E. To correctly reflect this source insertion in Figure 3(b), we simply add a new object via label E to object 17. But to reflect the same insertion in the minimal DataGuide in Figure 3(c), we must do more work in order to somehow generate the same DataGuide as our updated version of Figure 3(b), since it now is the minimal DataGuide for the source. In general, maintaining a minimal DataGuide in response to a source update may require much of the original database to be reexamined. The next subsection describes a second significant problem with minimal DataGuides.

2.5 Annotations

Beyond using a DataGuide to summarize the structure of a source, we may wish to keep additional information in a DataGuide. For example, consider a source with a label path *l*. To aid query formulation, we might want to present to a user sample database values that are reachable via *l*. (Such a feature is very useful in OEM, since there are no constraints on the type or format of atomic data.) As another example, we may wish to provide the user or the query processor with the statistical odds than an object reachable via *l* has any outgoing edges with a specific label. Finally, for query processing, direct access through the DataGuide to all objects reachable via *l* can be very useful, as will be seen in Section 6. The following definition classifies all of these examples.

Definition 6. In a source database *s*, given a label path *l*, a property of the set of objects that comprise the target set of *l* in *s* is said to be an *annotation* of *l*. That is, an annotation of a label path is a statement about the set of objects in the database reachable by that path. ❑

A DataGuide guarantees that each source label path *l* reaches exactly one object *o* in the DataGuide. Object *o* seems like an ideal place to store annotations for *l*, since we can access all annotations of *l* simply by traversing the DataGuide's single data path instance of *l*. Unfortunately, nothing in our definition of a DataGuide prevents multiple label paths from reaching the same object in a DataGuide, even if the label paths have different target sets in the source. Referring to Figure 3(c), we see that label paths A.C and B.C both reach the same object. Thus, if we store an annotation on object 20, we cannot know if the annotation applies to label path A.C, label path B.C, or both. In the DataGuide in Figure 3(b), however, we have two distinct objects for the two label paths, so we can correctly separate the annotations. Next, we formalize DataGuide characteristics that enable unambiguous annotation storage.

2.6 Strong DataGuides

We define a class of DataGuides that supports annotations as described in the previous subsection. Intuitively, we are interested in DataGuides where each set of label paths that share the same (singleton) target set in the DataGuide is exactly the set of label paths that share the same target set in the source. Formally:

Definition 7. Consider OEM objects s and d, where d is a DataGuide for a source s. Given a label path l of s, let $T_s(l)$ be the target set of l in s, and let $T_d(l)$ be the (singleton) target set of l in d. Let $L_s(l) = \{m \mid T_s(m) = T_s(l)\}$. That is, $L_s(l)$ is the set of all label paths in s that share the same target set as l. Similarly, let $L_d(l) = \{m \mid T_d(m) = T_d(l)\}$. That is, $L_d(l)$ is the set of all label paths in d that share the same target set as l. If, for all label paths l of s, $L_s(l) = L_d(l)$, then d is a *strong* DataGuide for s. ❑

For example, Figure 3(c) is not a strong DataGuide for Figure 3(a). The source target set T_s(B.C) is {6, 7}, and the DataGuide target set T_d(B.C) is {20}. In the source, L_s(B.C) is {B.C}, since no other source label paths have the same target set. In the DataGuide, however, L_d(B.C) is {B.C, A.C}. Since L_s(B.C) $\neq$ L_d(B.C), the DataGuide is not strong. The reader may verify that Figure 3(b) is in fact a strong DataGuide.

Next, we prove that a strong DataGuide is sufficient for storage of annotations.

Theorem 1. Suppose d is a strong DataGuide for a source s. If an annotation p of some label path l is stored on the object o reachable via l in d, then p describes the target set in s of each label path that reaches o.

Proof. Suppose otherwise. Then there exists some label path m that reaches o, such that p incorrectly describes the target set of m in s. This implies that $T_s(l) \neq T_s(m)$, since we know by Definition 6 that p is a valid property of $T_s(l)$. We reuse the notation from the definition of a strong DataGuide: let $L_d(l)$ denote the set of label paths in d whose target set is $T_d(l)$, and let $L_s(l)$ denote the set of label paths in s whose target set is $T_s(l)$. By construction, $L_d(l)$ contains both l and m. By definition of a strong DataGuide, $L_d(l) = L_s(l)$. Therefore l and m are both elements of $L_s(l)$. But this means that $T_s(m)$, the target set of m in s, is equal to $T_s(l)$, a contradiction to $T_s(l) \neq T_s(m)$, derived above. ❑

We also prove that a strong DataGuide induces a straightforward one-to-one correspondence between source target sets and DataGuide objects. This property is useful for incremental maintenance (Section 4) and query processing (Section 6).

Theorem 2. Suppose d is a strong DataGuide for a source s. Given any target set t of s, t is by definition the target set of some label path l. Compute $T_d(l)$, the target set of l in d, which has a single element o. Let F describe this procedure, which takes a source target set as input and yields a DataGuide object as output. In a strong DataGuide, F induces a one-to-one correspondence between source target sets and DataGuide objects.

Proof. We show that F is (1) a function, (2) one-to-one, and (3) onto. (1) To show F is a function we prove that for any two source target sets t and u, if $t = u$ then $F(t) = F(u)$. t is the target set of some label path l, and u is the target set of some label path m, so $t = T_s(l)$ and $u = T_s(m)$. If $t = u$, then l and m are both elements of $L_s(l)$, the set of label paths in s that share $T_s(l)$. Since d is strong, $L_s(l) = L_d(l)$. Therefore m is also an element of $L_d(l)$, $T_d(l) = T_d(m)$, and their single elements are equal. Hence $F(t) = F(u)$. (2) We show that F is one-to-one using the same notation and a symmetrical argument. If $F(t) = F(u)$, by construction we know that $T_d(l) = T_d(m)$. l and m are therefore both elements of $L_d(l)$, and by definition of a strong DataGuide are also elements of $L_s(l)$. Therefore $T_s(l) = T_s(m)$, i.e., $t = u$. (3) Finally, we see that the *accuracy* constraint of any DataGuide (Section 2.3) guarantees that F is onto. Any object in d must be reachable by some label path l that also exists (and therefore has a target set) in s. ❑

```
// MakeDataGuide: algorithm to build a strong DataGuide over a source database
// Input: o, the oid of the root of a source database
// Effect: dg is set to be the root of a strong DataGuide for o

targetHash = global empty hash table, to map source target sets to DataGuide objects
dg         = global oid

MakeDataGuide(o) {
    dg = NewObject()
    targetHash.Insert({o}, dg)
    RecursiveMake({o}, dg)
}

RecursiveMake(t1, d1) {
    p = set of <label, oid> children pairs of each object in t1
    foreach (unique label l in p)  {
        t2 = set of oids paired with l in p
        d2 = targetHash.Lookup(t2)
        if (d2 != nil) {
            add an edge from d1 to d2 with label l
        } else {
            d2 = NewObject()
            targetHash.Insert(t2, d2)
            add an edge from d1 to d2 with label l
            RecursiveMake(t2, d2)
        }
    }
}
```

Figure 4. Algorithm to create a strong DataGuide

If a DataGuide is not strong, it may be impossible to find a one-to-one correspondence between source target sets and DataGuide objects. For example, Figure 3(a) contains seven different target sets, each corresponding to one of the label paths A, A.C, A.C.D, B, B.C, B.C.D, and the empty path. Since Figure 3(c) has only 4 objects, we cannot have a one-to-one correspondence.

2.7 Building a Strong DataGuide

Strong DataGuides are easy to create. In a depth-first fashion, we examine the source target sets reachable by all possible label paths. Each time we encounter a new target set *t* for some path *l*, we create a new object *o* for *t* in the DataGuide—object *o* is the single element of the DataGuide target set of *l*. Theorem 2 guarantees that if we ever see *t* again via a different label path *m*, rather than creating a new DataGuide object we instead add an edge to the DataGuide such that *m* will also refer to *o*. A hash table mapping source target sets to DataGuide objects serves this purpose. The algorithm is specified in Figure 4. Note that we must create and insert DataGuide objects into `targetHash` before recursing, in order to prevent a cyclic OEM source from causing an infinite loop. Also, since we compute target sets to construct the DataGuide, we can easily augment the algorithm to store annotations in the DataGuide.

3. Experimental Performance

As described in Section 2.3, computing a DataGuide for a source is equivalent to converting a non-deterministic finite automaton into an equivalent deterministic finite automaton. For a tree-structured source, this conversion always runs in linear time, and the size of the DataGuide is bounded by the size of the source. Yet for an arbitrary graph-structured source, creating a DataGuide may require exponential running time and could feasibly generate a DataGuide exponentially larger than the source. Needless to say, we are very concerned about the potential for exponential behavior, and as far as we know no research has tried to formalize automaton characteristics that lead to better or worse behavior.

Source					DataGuide		
Description	Objects	Links	Labels	Height	Objects	Links	Time (secs)
Sports (Tree)	3,095	3,094	41	5	75	74	1.37
DBGroup (Graph)	947	1,102	32	--	138	168	1.52

Table 1. DataGuide performance for operational Lore databases

In this section, we show that for many classes of OEM databases, experimental performance results are very encouraging. We begin by discussing performance on two operational OEM databases that, although admittedly are relatively small, require very little time for DataGuide creation and yield DataGuides significantly smaller than the source. We then describe further experiments conducted on synthetic OEM databases. For a wide range of parameters, we find that many large graph-structured databases still yield good performance. All measurements are taken running the Lore system on a Sun Ultra 2 with 256MB RAM.

3.1 Operational Databases

We first consider two medium-sized databases used in Lore. One is a tree, and the other is a graph with significant data sharing. We believe tree-structured sources will be common in Lore; any relational database, for example, can be modeled as an OEM tree. Our tree-structured database contains a snapshot of data imported from a large and popular Web site covering many different sports, with the OEM database following the structure of the menus and links at the site. While the overall structure is quite regular, data for each sport differs significantly. We captured only a small portion of the Web site, building a database with about 3,000 objects and links, 40 unique labels, and a maximum height of 5. Building a strong DataGuide requires 1.37 seconds, and the DataGuide contains 75 objects and 74 links.

Our second operational database contains information about the Stanford Database Group, describing the group's members, projects, and publications. (We will see this database again in Section 5 when we discuss Lore's user interface.) The database uses extensive data-sharing (graph structure). As an example, a single group member might be reachable as a member of one or more projects and as an author of any number of publications. The graph also contains numerous cycles; for example, each group member reachable by a link from a project also has links to all projects he or she works on. Our database currently contains about 950 objects and 1,100 links, with 32 unique labels. Building a strong DataGuide takes 1.52 seconds; the resulting DataGuide has 138 objects and 168 links. Performance for both databases is summarized in Table 1.

3.2 Synthetic Databases

To further study performance, we generated numerous large synthetic databases, both trees and graphs, with and without cycles. For tree-structured databases we have the following parameters.

- *Height*, or number of levels, in the tree.
- For each level in the tree, the number of unique labels on outgoing edges (*labels per level*). The sets of labels corresponding to different levels are disjoint.
- Maximum number of outgoing edges from any non-leaf (*fan-out*).
- Whether to use maximum fan-out for each object (*full)* or to simulate irregular structure by varying the number of outgoing edges of any object from zero to the maximum fan-out (*irregular*).

For graph-structured databases we modify and supplement the above tree parameters as follows.

- *Height* is defined as the longest path in a breadth-first traversal from the root of the graph. Level n includes all objects whose shortest path from the root has n edges.
- *Fan-out* no longer is sufficient to specify the number of objects at a level, since many edges of one level may point to the same object. Hence, a new parameter is the maximum number of *objects per level*, as an integer to be multiplied by the level number. Until this number is exceeded, every edge

Source										DataGuide		
DB No	Tree ?	Objects	Links	Height	Labs. per Level	Fan-out	Full ?	Objs. per Level	Backlink Freq/ Level	Objects	Links	Time (secs)
1	Y	37,449	37,448	5	1	8	Y	--	--	6	5	11.3
2	Y	329,176	329,175	12	2	8	N	--	--	1,802	1,801	127.3
3	N	37,111	311,111	12	2	10	Y	500	--	156	288	123.1
4	N	26,700	93,151	12	2	10	N	500	--	3,074	3,073	712.6
5	N	11,134	44,346	5	4	80	N	2000	10/2	198	720	22.6
6	N	4,524	13,151	8	4	10	N	200	10/0	14,326	29,101	78.5
7	N	3,108	6,787	8	4	10	N	200	15/3	8,736	16,805	36.2

Table 2. DataGuide performance for synthetic databases

from the previous level points to a different object. When the limit is reached, all remaining edges are evenly distributed among existing objects in the level.

- Rather than sending all outgoing edges to objects in the next level, any proportion of outgoing edges (*backlink frequency*) may be redirected to objects in previous levels; here we always redirect edges to objects a fixed number of levels (*backlink level*) above the current level.

The results discussed below are captured in Table 2. We begin by summarizing the performance for two tree-structured databases. A large full tree with only one label per level provides an extreme example of how a DataGuide can be very small when compared to the source. *DB1*, a full tree with a fan-out of 8, height of 5, and one label per level, contains 37,449 objects. The strong DataGuide contains only 6 objects, and building it takes 11.3 seconds. As a larger example, we built *DB2*, which has an irregular edge distribution with a maximum fan-out of 8, height of 12, and 2 labels per level. The tree contains 329,176 objects. It takes 127.3 seconds to build a strong DataGuide with 1,802 objects.

Next, we describe several graph-structured databases. We begin with a regular, cycle-free graph, and then progress to more intricate examples. In *DB3*, each non-leaf has 10 outgoing edges, with two labels per level. There are 12 levels of objects, with a maximum of 500 objects in level 1, 1,000 in level 2, 1,500 in level 3, and so on. The source database has 37,111 objects and 311,111 links. The DataGuide has 156 objects and 288 links, requiring 123.1 seconds to create. Next, we introduce irregularity in the number of outgoing edges from each object. This irregular version, *DB4*, is expectedly smaller, with 26,700 objects and 93,151 links. The irregularity results in more time for DataGuide creation and a larger DataGuide: 712.6 seconds, with 3,074 objects and 3,073 links.

For the remaining databases we introduce backlinks, which clearly can complicate DataGuide performance. We begin with *DB5*, which has relatively shallow height (5) but large breadth, with 80 outgoing edges per object and up to 2,000 objects on level 1, 4,000 on level 2, etc. Every tenth edge is a backlink to an object two levels closer to the root. The database has 11,134 objects and 44,346 links, and it yields good performance: 22.6 seconds to build the DataGuide, which has 198 objects and 720 links. In practice, we expect many databases to follow this style, generally structured as a wide but reasonably shallow tree with some cycles and links for data-sharing.

For our next examples, we reduce the breadth and significantly increase the height; we cut fan-out to 10, reduce objects per level to at most 200 times the level number, and increase height to 12. In *DB6*, we make every tenth edge a link to another object at the same level. While the time required to create the DataGuide is still reasonable, we see that the DataGuide has become larger than the source. Keep in mind that even if larger than the source, the properties of any strong DataGuide make it useful for schema browsing and query optimization, as we will discuss later. In *DB7*, we have fewer backlinks but allow them to point to objects three levels closer to the root. Performance is similar, with fast creation time but a DataGuide larger than the source.

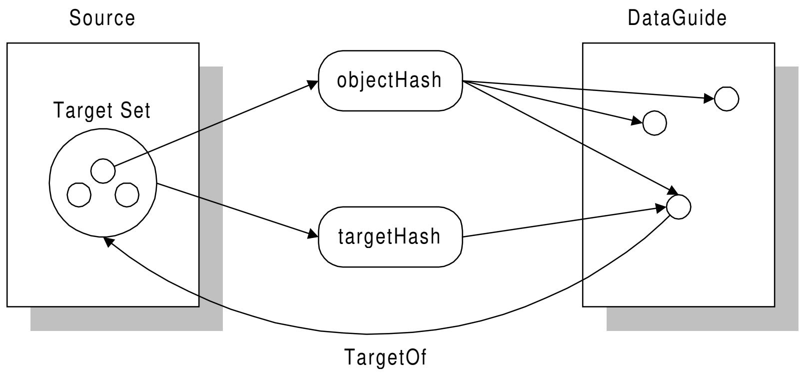

Figure 5. Data structures to support DataGuide maintenance

While it is impossible here to explore all possible graphs, our results categorize performance for a significant range of databases. In summary, we see that as expected, performance for any tree is good. Acyclic graphs with repetitive structure do not cause problems in common situations. For relatively shallow graphs with a large number of outgoing edges per object, cycles do not pose much of a problem either. For much deeper graphs, however, cycles can cause DataGuides to be larger than the source. While the examples presented here yield reasonable performance, the potential does certainly exist for very poor performance. Many unconstrained backlinks in deep graphs, for instance, can cause significant problems. While we are confident that in practice OEM databases will rarely exhibit structure that results in poor performance, we plan to continue to investigate the matter by building additional operational Lore databases for empirical testing. Also, we hope to formalize properties that can guarantee (or prohibit) good performance, or to find heuristics to help an algorithm detect when a database may result in poor performance. In such cases, we may be able to achieve better performance by building a strong DataGuide over only the first few levels. This way, DataGuides can still be useful for guiding queries that do not examine long paths.

4. Incremental Maintenance

If a DataGuide is to be useful for query formulation and especially optimization, we must keep it consistent when the source database changes. In this section we address how to update a strong DataGuide to reflect insertions or deletions of edges in the source. Note that updates to atomic values do not affect the DataGuide. We modify the DataGuide creation algorithm in Figure 4 for incremental maintenance. First, we list changes to the algorithm's data structures, as summarized in Figure 5.

- As we construct target sets in the DataGuide algorithm (in variables `t1` and `t2`), we store them within the database as auxiliary OEM objects.
- We make persistent the `targetHash` table, which maps source target sets to DataGuide objects.
- For each DataGuide object, we add an edge connecting it to its corresponding target set (guaranteed to exist by Theorem 2). The edge has the special label `TargetOf`.
- In parallel, we build an additional persistent hash table, `objectHash`, to map a source object *o* to all DataGuide objects that correspond to target sets containing *o*.

```
    // Algorithm to update a DataGuide in response to source insertions or deletions
    // Input: U, a set of edge updates, each of the form u.l.v
    // Effect: The global DataGuide dg correctly reflects all updates to the source

    targetHash = global persistent hash table, mapping source target sets to DataGuide objects
    objectHash = global persistent hash table, mapping source objects to DataGuide objects
    dg         = global oid of the root of a strong DataGuide

    HandleUpdate(U) {
        foreach (update point u in U) {
            foreach (DataGuide object d in objectHash.Lookup(u)) {
                RecursiveMake(TargetOf(d),d)
            }
        }
    }

    RecursiveMake(t1, d1) {
        p = set of <label, oid> children pairs of each object in t1
        foreach (unique label l in p)  {
            t2 = set of oids paired with l in p
            d2 = targetHash.Lookup(t2)
            if (d2 != nil) {
(1)             if an edge does not already exist from d1 to d2 with label l {
(2)                 if d1 has an outgoing edge with label l, remove it
                    add an edge from d1 to d2 with label l
(3)             }
            } else {
                d2 = NewObject()
                targetHash.Insert(t2, d2)
(4)             foreach (oid o in t2) {
(5)                     objectHash.Append(o, d2)
(6)             }
(7)             TargetOf(d2) = t2
(8)             if d1 has an outgoing edge with label l, remove it
                add an edge from d1 to d2 with label l
                RecursiveMake(t2, d2)
            }
        }
(9)     remove any outgoing edges of d1 (other than TargetOf) with a label not in p
    }
```

Figure 6. Algorithm to incrementally update a strong DataGuide

Our algorithm updates the DataGuide in response to any number of edge insertions or deletions on the source. Each edge can be written as *u.l.v*, indicating an edge from object *u* to object *v* via the label *l*. We refer to *u* as the *update point*. (When adding an edge, *v* may or may not already exist in the database.) Note that the algorithm can directly handle the insertion of a complete subgraph, given an update point connecting the new graph to the existing database. The first step of the algorithm is to identify all DataGuide regions that might be affected by the changes: for each update point *u*, we use `objectHash` to find every DataGuide object whose corresponding source target set contains *u*. Each such DataGuide object is a "sub-DataGuide" that describes the potential structure of any object in the corresponding source target set (including one or more of the update points). The updates may affect each such sub-DataGuide, so we must reexamine all of them, relying on `targetHash` to avoid excessive recomputation. The algorithm turns out to be only a slightly modified version of the DataGuide creation algorithm from Figure 4. In fact, the new `RecursiveMake` algorithm can and should be used to build the initial DataGuide to ensure that the data structures are built correctly. The algorithm is presented in Figure 6. Lines that are different from the original `RecursiveMake` algorithm are numbered and emphasized.

The `HandleUpdate` algorithm is very simple, using `objectHash` to identify all sub-DataGuide objects that might need to be updated. The modifications to `RecursiveMake` are as follows. Line (1) checks to

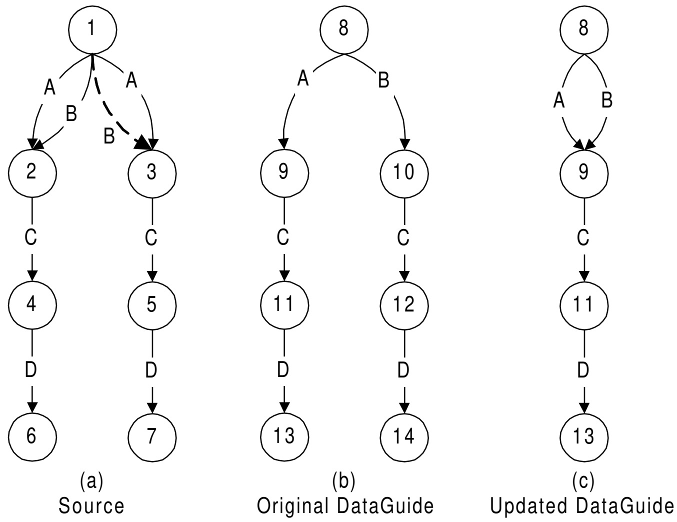

Figure 7. Insertion of an edge

make sure that the exact edge we wish to add does not already exist. In truth this check is only an optimization, since the two lines following the check would simply remove and re-add that edge. Line (2) removes old DataGuide edges that are no longer correct: a change in target sets may cause a DataGuide edge to point to a new object. Lines (4)-(7) simply maintain `objectHash` and the `TargetOf` links when new objects are added to the DataGuide.[1] Line (8) performs the same function as line (2). To preserve DataGuide accuracy, line (9) removes DataGuide edges with labels no longer represented in the source due to edge deletion. The edge removal in lines (2), (8), and (9) may result in detached subgraphs in the DataGuide. In Lore, garbage collection periodically deletes any unreachable objects. We must at the same time remove obsolete references from the persistent hash tables.

Next, we trace two examples to demonstrate the algorithm.

Example 4.1. Figure 7 shows one of the trickier cases for insertion. Figure 7(a), without the dashed B edge between objects 1 and 3, is our original source, and Figure 7(b) is a strong DataGuide for this source (with `TargetOf` links omitted). Suppose we insert the B edge. `HandleUpdate` is called with the argument {1.B.3}, and 1 is the sole update point. DataGuide object 8 corresponds to the only target set that object 1 is a part of. Hence, we call `RecursiveMake` with {1} as the initial target set and 8 as the initial DataGuide object. As in the original algorithm, we examine the children of all objects in the initial source target set, label by label. Suppose we consider children via label A first. The target set `t2` is {2, 3}. From our persistent `targetHash`, we see that object 9 corresponds to this set. Line (1) catches the fact that an edge from 8 to 9 with the label A already exists, so no additional work is required for that label. Proceeding to examine children via label B, we see that the target set is now also {2, 3}. Hence we add a new edge from 8 to 9 with the label B. Before doing so, we remove the existing B edge, as specified by line (2) in `RecursiveMake`. The detached subgraph is garbage collected, and the final result is the strong DataGuide shown in Figure 7(c).

Notice that deleting the edge we just inserted would regenerate a DataGuide equivalent to Figure 7(b). After the deletion, the target set of A remains {2, 3}, but the target set of B is now {2}. Hence, the B edge from 8 to 9 is removed, and recursive calls to `RecursiveMake` generate a new DataGuide path from the root for B.C.D. ❑

Example 4.2. We now demonstrate how the algorithm handles deletion in a case where we must recompute multiple sub-DataGuides. Figure 8(a), including the dashed E edge from 6 to 9, is our source. Note that object 6 is in two target sets, {5, 6} for A.C, and {6, 7} for B.C. Figure 8(b) is the original strong DataGuide. Suppose we delete the E edge. Because object 6 is in two target sets, we must reconsider two sub-DataGuides, objects 14

[1] Similar lines also must be added to the `MakeDataGuide` function in Figure 4 to correctly store root information.

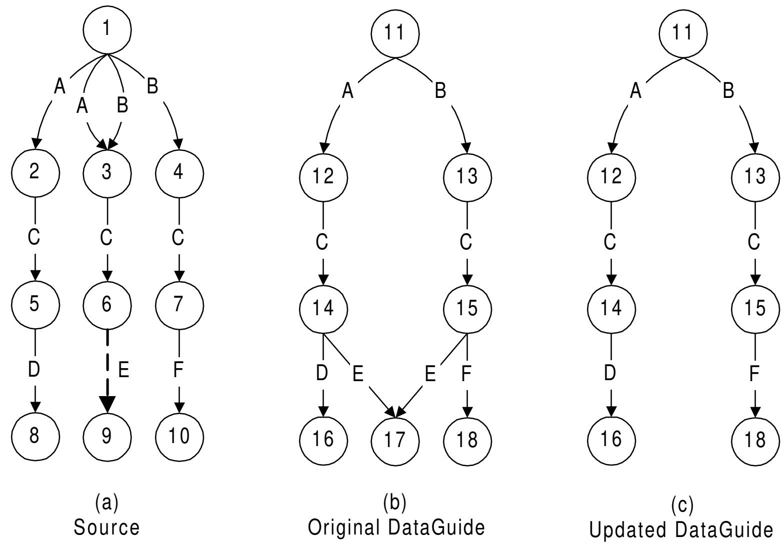

Figure 8. Deletion of an edge

and 15. Consider 14 first. We call `RecursiveMake` with target set {5, 6} and object 14 as arguments. The target set for children via label D is {8}, which already corresponds to object 16, so no change is made. There are no other children to consider, and line (9) of the algorithm will remove the obsolete E edge from object 14. Calling `RecursiveMake` for target set {6, 7} and object 15, we eliminate the other E edge in the same manner, and object 17 is garbage collected. The final result is in Figure 8(c). ❑

The work required to maintain the DataGuide depends entirely on the structural impact of the updates. For example, inserting a new leaf into a tree-structured database requires only one target set to be recomputed (and one new object added to the DataGuide). At the other extreme, in a graph-structured databases extensive sharing may cause many sub-DataGuides to be recomputed after an update. Regardless, keeping accurate target set data prevents any excessive recomputation: recursion is halted whenever a target set lookup in `targetHash` is successful, indicating that the sub-DataGuide corresponding to that target set is already correct.

5. Query Formulation

Without some notion of the structure of a database, formulating queries can be extremely difficult. The user is limited to an ad-hoc combination of browsing the entire database, issuing exploratory queries, and guesswork. Since DataGuides provide concise, accurate, and up-to-date summarizing information about the structure of a database, they are very useful for query formulation. In this section we demonstrate the value of DataGuides in the context of a Java-based Web user interface we have created for Lore. From the interface, a user can interactively explore the DataGuide to aid formulation of Lorel queries. Further, the DataGuide enables end-users to specify a large class of queries in a "by example" style, without any knowledge of the Lorel query language.

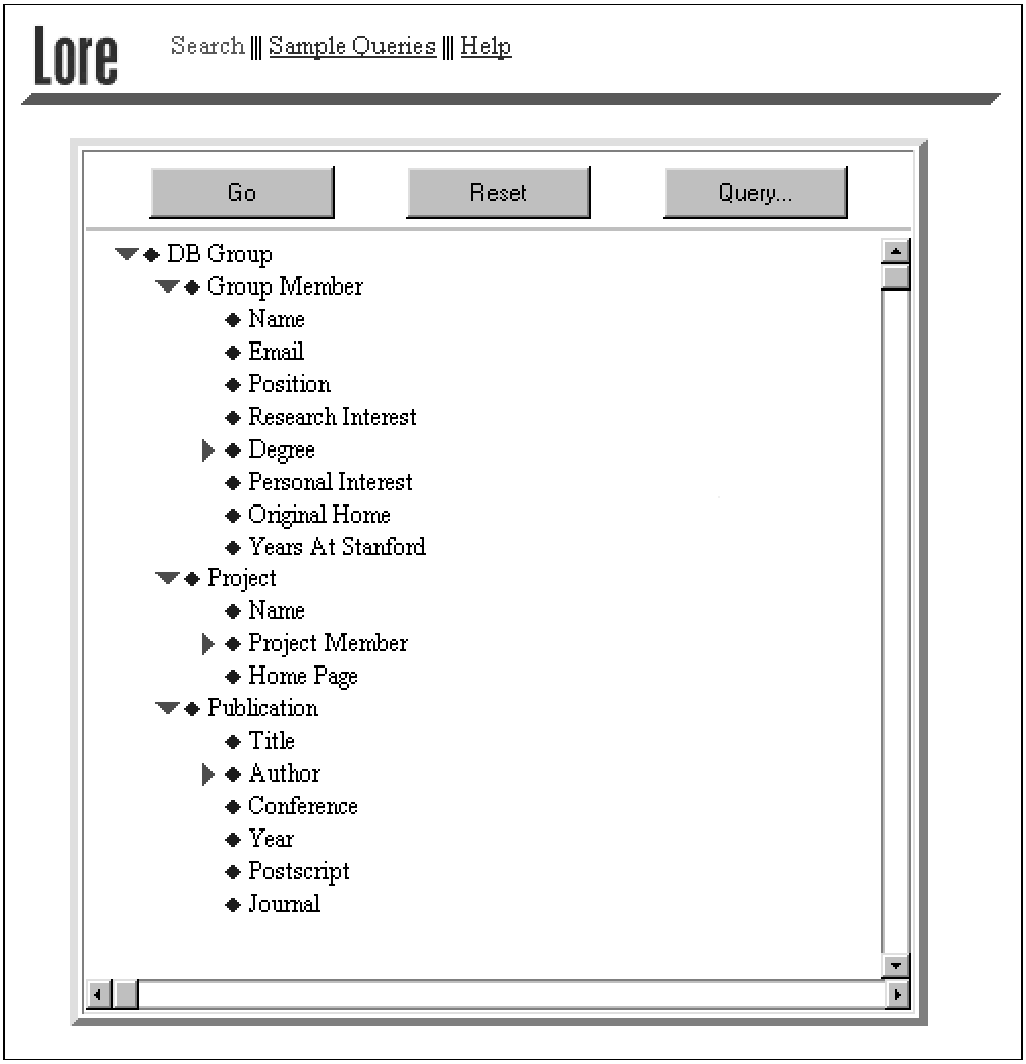

Figure 9. A Java DataGuide

In all of our examples we refer to a medium-sized database we have built describing members, projects, and publications of the Stanford Database Group, first introduced in Section 3. The database mirrors much of the information available on the Database Group Web site, and in fact contains links to many of our site's home pages, images, and publications. Once a connection to the database is made, the user is presented with an HTML page framing a Java DataGuide, as shown in Figure 9.

The user can explore the DataGuide by clicking on the arrows (triangles), which expand or collapse complex objects within the DataGuide. Immediately, we see how the DataGuide guides the specification of path expressions used in queries (recall Section 2.2): every valid path expression must begin with the `DB_Group` label, followed by `Group_Member`, `Project`, or `Publication`. Expanding a DataGuide complex object lists all potential subobject labels that are found in the database, and we never see two subobjects with the same label. Therefore, we can determine whether any label path of length n exists in the database by clicking on at most n-1 DataGuide arrows. In contrast, when browsing a semistructured database directly, we may have to examine many like-labeled objects before finding one with a specific outgoing label.

While the DataGuide is useful for deducing valid path expressions, values in the database at this point remain a mystery. A user interested in locating all group members from Nevada doesn't know if `Original_Home` for someone from Las Vegas would be stored as `"Las Vegas, NV"`, `"Nevada"`, or `"Nevada, USA"`. One option is to use Lorel's pattern matching features [AQM+96] to write a query that attempts to encompass all possible formats, but in many cases a better approach is to examine sample values

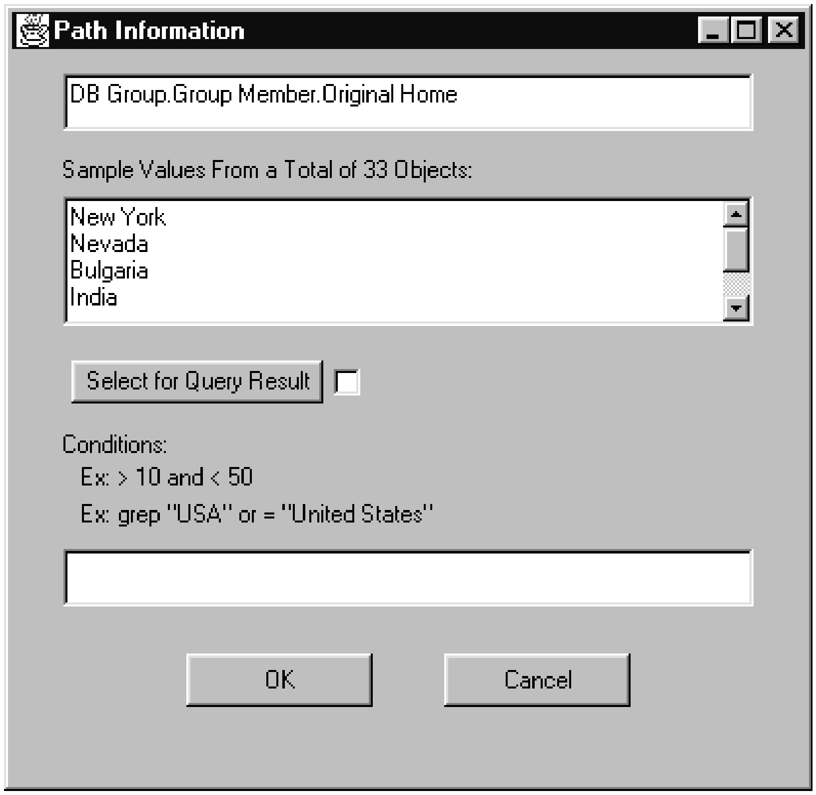

Figure 10. DataGuide path information

from the database. As described in Section 2.5, we can effectively store such sample values as annotations in the DataGuide. In Figure 9, notice that a diamond accompanies every label, corresponding to a distinct label path from the root. Clicking on the diamond brings up a dialog box such as the one shown in Figure 10, which was obtained by clicking on the diamond next to the `Original_Home` label.

The top portion of the dialog box identifies the path expression and shows two DataGuide annotations: the total number of database objects reachable by that path expression, and a list of sample values. Currently, a fixed number of values are chosen arbitrarily from the database, although clearly we could be more sophisticated here. Annotations are stored as specially marked children of DataGuide objects that are interpreted by the user interface. They are computed during DataGuide creation and maintenance by simple extensions to the algorithm in Figure 6.

The other elements in the dialog box allow users to specify queries directly from the DataGuide without writing Lorel, in a style reminiscent of *Query By Example* [Zlo77]. As shown, a user can click a button to select a path for the query result. Further, value-filtering conditions may be specified using common arithmetic and logical operators, as well as custom operators such as the UNIX utility `grep` and the SQL function `like`. (These comparisons correspond to Lorel "where" conditions, but users need not be aware of that fact.) The on-screen DataGuide is updated to reflect any query specifications, highlighting diamonds for selected path expressions and displaying filtering conditions next to the corresponding label. Figure 11 shows the DataGuide after a user has specified to select all graduate students in the group that are originally from Nevada or New York and have been at Stanford for more than two years. (The `like` predicate will satisfy any `PhD Student` or `Masters Student`.) When the user clicks the `Go` button from Figure 9, the Java program generates a Lorel query equivalent to the DataGuide query specification, and sends it to Lore to be processed. Lore returns query results in HTML, using a hierarchical format that is easy to browse and navigate: like-labeled objects are grouped together, and complex objects are represented as hyperlinks. At any point the user may return to the DataGuide to modify the original query or submit a new one.

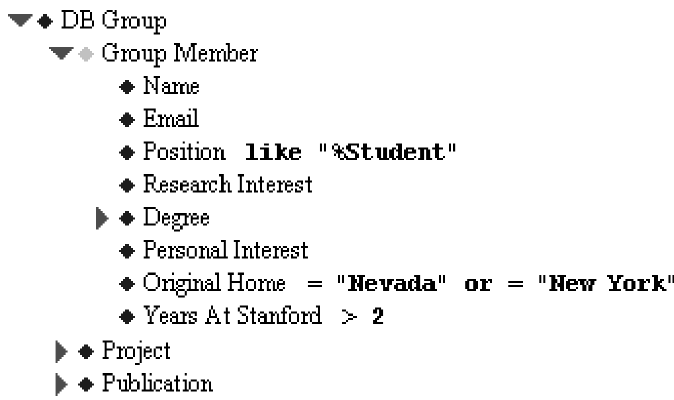

Figure 11. A DataGuide query specification

Currently, DataGuide queries can specify any Lorel query with simple path expressions (no path wildcards) and "where" clauses that are conjunctive with respect to unique path expressions. Also, all value comparisons must be made against constants. We hope to add techniques that expand the expressive power of DataGuide queries; e.g., disjunctions across path expressions, path wildcard specifications, and variables to enable joins.

On a larger scale, we believe that there is much opportunity for blurring the distinction between formulating a query and browsing a query result, in the spirit of PESTO [CHMW96]. For example, suppose that instead of supplying just a few sample values, the dialog box for each path expression always displayed all values. Then clicking on a diamond answers the simple query to find all values reachable by a given path. Furthermore, by integrating the query processor with our DataGuide maintenance algorithms, we could quickly respond to a filtering condition specified in the DataGuide by updating the DataGuide and its value lists to reflect that condition. For example, suppose the user specified the condition in Figure 11 on `Position` first, restricting the query to only consider students. It may be that the database has no `Research_Interest` data for any such group members, so that path could be removed temporarily from the DataGuide. More importantly, clicking on the diamond next to `Original_Home` would now display the homes of students only. In the same manner, restricting `Years_At_Stanford` would evaluate the entire desired query, since clicking on the diamonds for labels under `Group_Member` would only display data that matched our query conditions. At that point, it may be desirable to revert to the current model of result browsing, allowing a user to examine one by one the group members that satisfied the query.

The DataGuide-driven user interface described here is accessible to the public via the Lore Home page on the Web, at `www-db.stanford.edu/lore`.

6. Query Optimization

In this section we discuss how the information maintained by a strong DataGuide can be used to significantly speed up query processing for a broad class of Lorel queries. Essentially, a strong DataGuide can also serve as a *path index*. While path indexes have been studied for traditional object-oriented systems, e.g., [BK89, CCY94, KM92], their use in a semistructured environment has not been addressed. In particular, creating and maintaining a path index without a fixed schema may be quite difficult, yet we can conveniently use strong DataGuides to address the problem. As shown in Section 4 for incremental maintenance, each object in the strong DataGuide can have a link to its corresponding target set in the source. Hence, in time proportional to the length of a label path, we can use the DataGuide to find all source objects reachable via that path, independent of the size of the source. In this section we analyze a sequence of queries to show the benefits of having fast access to target sets during query processing.

All of our query processing comparisons are based on the number of objects examined. We use a very simple cost model that assigns a uniform cost to every object examination since, in general, it is difficult to make guarantees about clustering in a graph-based model like OEM; each object examination may therefore require a random disk access. Note that the value of a complex object is a sequence of <label, oid> pairs representing its subobjects [MAG+97], so time spent to examine only the labels and oids of those subobjects is included in the cost of examining the complex object itself. For some queries, we need to find parents of an OEM object. Parent pointers need not be stored explicitly within the database; Lore, for example, instead uses a hash-based index to map an object *o* and a label *l* to all parents that reach *o* via *l* [MAG+97]. For simplicity, we assume that examining an object yields that object's parents at no additional cost.

Example 6.1. We begin by tracing a very simple Lorel query over a sample database, showing how the DataGuide can dramatically reduce query execution cost. Suppose we wish to execute the following Lorel query (recall Section 2.2) over a database with structure similar to the Stanford Database Group database described in Section 5. It finds all publications in Troff format.

```
Select DB_Group.Group_Member.Publication.Troff
```

The result is a set of oids. For this example, let us consider an extreme database that has one DBGroup object containing 10,000 group members (among other objects). Each GroupMember has an average of 100 Publications, but only one Troff subobject exists in the entire database. Without any a priori knowledge of the structure of the database, a query processor would be forced to examine each GroupMember, in turn each Publication of each GroupMember, and finally return every Troff object of each such Publication. We see that, in addition to the root and the DBGroup object, the query processor must examine 1,000,000 objects. Note that Lore's current indexing schemes are not applicable to this query [MAG+97].

In this example, the query result is exactly the objects in the target set of DBGroup.GroupMember.Publication.Troff. To find the target set, we simply traverse the path from the root of the DataGuide, and we know there is only one such path. Hence, we need examine only six objects to find the result: the DataGuide root, the DBGroup object, the GroupMember, the Publication, the Troff object, and the object containing the path's target set. (As in Section 4, the object in the DataGuide reachable by DBGroup.GroupMember.Publication.Troff includes as part of its value a TargetOf link to a special complex object whose children are all objects in the path's target set.)

Note that when traversing the DataGuide, we may find that a path does not exist. For this query and many others, such a finding guarantees that the query result is empty. This type of optimization does not require a strong DataGuide and was in fact suggested by [NUWC97]. ❑

Example 6.2. We now show a somewhat more interesting query. Suppose we wish to find the publication years of some of the group's older publications:

```
Select DB_Group.Group_Member.Publication.Year
Where  DB_Group.Group_Member.Publication.Year < 1975
```

This query is similar to the previous example but introduces a filtering condition. For such conditions Lore includes a B-tree based *value index* (*Vindex*) that takes a label, operator, and value and returns the set of oids of objects that satisfy the given value constraint and have the specified incoming label [MAG+97]. Note that this index is based only on the last label in a label path to an object. Using the DataGuide, we can compute the intersection between the set of objects returned by the Vindex on (Year, <, 1975) and the target set of the full label path, DBGroup.GroupMember.Publication.Year. Because the DataGuide algorithm in Figure 6 constructs each target set in one step (and never modifies a target set), we can typically expect target sets to be stored contiguously on disk. Further, since oids returned by the Vindex are stored efficiently in a B-tree, we expect computation of this intersection to be fast, with few additional random disk accesses.

We now specify a sample database for analyzing the performance of both this query and Example 6.3 below. While the numbers are contrived in this particular database, they are representative of the size and structure of databases we are likely to encounter in practice. Suppose the path DBGroup.GroupMember.Publication.Year has a target set *Y* of 20,000 objects. Assume 1,000 of these objects satisfy the value constraint, each reachable via a single Publication along that path. Also, suppose that these

1,000 Year objects are referenced by 1,000 other Publications along the path DBGroup.Project.Publication.Year, and that 9,000 other Year objects with value less than 1975 are reachable from 9,000 more Publications on that same path. Hence, a Vindex lookup on (Year, <, 1975) returns 10,000 objects, pointed to by 11,000 different Publications.

To process the query using the DataGuide, we first examine 5 DataGuide objects to find the oid identifying *Y*. Next, we retrieve the 10,000 valid oids from the Vindex and intersect them with the 20,000 oids of *Y* to compute the result. Now consider processing the query without the DataGuide. A "top-down" exploration that does not use the Vindex would need to examine the values of all 20,000 objects in *Y*, and as in the previous example we might examine many GroupMember or Publication objects that do not even have the appropriate subobjects. Alternatively, Lore can build a query plan to take advantage of the Vindex by traversing "bottom-up" to identify objects reachable by valid paths [MAG+97]. In this example, for each object *o* returned by the Vindex, the system would find all objects that have a Year link to *o*, check to see which of those objects have incoming links with the label Publication, and so on up to the root until it can determine whether or not the object is indeed reachable via the label path DBGroup.GroupMember.Publication.Year. To begin processing our example, we first examine all 10,000 objects returned by the Vindex to find the 11,000 Publications with links to those objects. Next, we must find the parents of all 11,000 Publication objects as well. Hence, processing the query "bottom-up" requires at least 21,000 objects to be examined. ❑

Example 6.3. Suppose we now wish to find the actual older publications:

```
Select DB_Group.Group_Member.Publication
Where  DB_Group.Group_Member.Publication.Year < 1975
```

Let *P* denote the target set of the "select" path and *Y* the target set of the "where" path, both found by traversing a single data path in the DataGuide. As mentioned in Example 6.1, if either path does not exist then the query result is empty. Otherwise, we proceed as in Example 6.2 to intersect oids in *Y* with the set of oids returned by the Vindex to identify candidate Year objects, *Y**. Next, we examine all objects in *Y** to find the set *P** of (parent) objects that have Year links to objects in *Y**. Since *P** may include objects not in the query result, we intersect the oids of *P** and *P* to compute the final result *R*.

As before, *Y* has 20,000 objects. We assume each Publication has a single Year, so *P* has 20,000 objects as well. *Y**, essentially the query result from the previous example, has 1,000 objects. Because of data-sharing, *P** contains 2,000 objects. In addition to the work required from the previous example to compute *Y**, we need to examine the 1,000 objects in *Y** to find the parent objects in *P**, and we must intersect *P* and *P** to find *R*. Hence, the total cost using the DataGuide is 1,000 expensive object examinations, plus the relatively small costs involved in retrieving 10,000 oids from the Vindex and performing two oid set intersections: one between the 10,000 oids returned by the Vindex and the 20,000 oids in *Y*, and the other between the 20,000 oids in *P* and the 2,000 oids in *P**. In comparison, a top-down approach without the Vindex or DataGuide would again have to examine at least 20,000 objects. Similarly, as in the previous example, combining the Vindex with parent traversal would retrieve 10,000 oids from the Vindex and then examine at least 21,000 objects. ❑

The three examples illustrate how the DataGuide can be used to significantly speed up common queries. These techniques can be generalized to many other queries as well. Since DataGuides are stored as OEM objects we can also optimize queries with more sophisticated path expressions: Lorel supports "wildcards" and regular expressions in path specifications [AQM+96]. For example,

```
Select DB_Group(.Group_Member | .Project).Publication
```

selects Publications of either GroupMembers or Projects. Because the DataGuide is an OEM object, we can reuse the same code that handles such constructs over data to find target sets of such paths in the DataGuide.

In practice, the impact of the DataGuide on query processing certainly depends on the structure of the database. Even so, direct access to target sets always enables the query processor to prevent the search space from growing needlessly large. As follow-on work, we plan to run benchmarks to carefully compare the performance of the different query processing approaches described in this section. Ultimately, we hope to build an optimizer that uses statistical knowledge of a database and detailed performance characteristics to combine DataGuides, Vindexes, and child/parent link traversal into efficient query plans.

7. Conclusion and Future Work

We have presented DataGuides, a novel feature designed to support various aspects of managing and querying semistructured data. Rather than require an explicit schema that all data must follow, a DBMS can support free-form data, dynamically generating and maintaining a DataGuide that summarizes the structure of the data. DataGuides provide key benefits afforded by a schema, such as guidance to the user for query formulation and guidance to the query processor for query optimization. After setting the stage with formalization and algorithms, we focused on experimental and practical application of DataGuide technology in Lore, a DBMS for semistructured data. We have found that for typical databases DataGuides are easy and fast to create. DataGuides were shown to be an important part of Lore's user interface, and we also explained how a DataGuide can be used to significantly speed up query processing.

We are considering the following areas for future research.

- From a theoretical standpoint, we would like to investigate the possibility of performance guarantees for DataGuide creation over certain classes of databases. Ideally, we could formalize database characteristics that guarantee good performance. Heuristics that quickly identify databases that may result in poor DataGuide performance would also be helpful. Strategies for dealing with such cases are also important.
- As mentioned in Section 5, we plan to continue to exploit DataGuides to enhance our user interface to Lore. In addition to allowing more expressive queries to be specified directly from the DataGuide, we plan to work towards blurring the distinctions between metadata and data (or alternatively, query formulation and result browsing). This process will demand considerable cooperation between the query processor and DataGuide management, in addition to quickly and repeatedly updating a (potentially remote) user's view of the database.
- With regard to query optimization, we plan to run extensive benchmarks comparing query processing in Lore with and without DataGuides. In the process, we seek to classify the queries and database characteristics for which DataGuides improve performance.

Acknowledgments

The authors wish to thank Svetlozar Nestorov, Jeff Ullman, Janet Wiener, and Sudarshan Chawathe for their initial work on Representative Objects. We are also grateful to Serge Abiteboul, Jason McHugh, Svetlozar Nestorov, and Jeff Ullman for helpful suggestions on our work, and to the rest of the Lore group at Stanford for enabling this research.

References

[AGS90] R. Agrawal, N. Gehani, and J. Srinivasan. OdeView: The Graphical Interface to Ode. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 34-43, Atlantic City, NJ, May, 1990.

[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.

[BDFS97] P. Buneman, S. Davidson, M. Fernandez, and D. Suciu. Adding Structure to Unstructured Data. In *Database Theory: Sixth International Conference Proceedings*, pp. 336-350, Delphi, Greece, 1997.

[BDHS96] P. Buneman, S. Davidson, G. Hillebrand, and D. Suciu. A Query Language and Optimization Techniques for Unstructured Data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 505-516, Montreal, Canada, 1996.

[BDS95] P. Buneman, S. Davidson, and D. Suciu. Programming Constructs for Unstructured Data. In *Proceedings of the 1995 International Workshop on Database Programming Languages*, 1995.

[BK89] E. Bertino and W. Kim. Indexing Techniques for Queries on Nested Objects. *IEEE Transactions on Knowledge and Data Engineering*, pp. 196-214, 1(2), June, 1989.

[Cat93] R.G.G. Cattell, ed. *The Object Database Standard: ODMG-93*. Morgan Kaufmann, San Francisco, California, 1994.

[CCY94] S. Chawathe, M. Chen, and P. Yu. On Index Selection Schemes for Nested Object Hierarchies. In *Proceedings of the Twenty-First International Conference on Very Large Data Bases*, pp. 331-341, Santiago, Chile, 1994.

[CHMW96] M. Carey, L. Haas, V. Maganty, and J. Williams. PESTO: An Integrated Query/Browser for Object Databases. In *Proceedings of the Twenty-Second International Conference on Very Large Data Bases*, pp. 203-214, Bombay, India, August, 1996.

[Hop71] J. Hopcroft. An n log n Algorithm for Minimizing the States in a Finite Automaton. In *The Theory of Machines and Computations*, pp. 189-196, New York, 1971.

[HU79] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, Massachusetts, Date, 1979.

[KM92] A. Kemper and G. Moerkotte. Access Support Relations: An Indexing Method for Object Bases. *Information Systems*, pp. 117-145, 17(2), 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*, pp. 54-65, Zurich, Switzerland, 1995.

[MAG+97] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A Database Management System for Semistructured Data. Technical Report, Stanford University Database Group, February, 1997.

[MDT88] A. Motro, A. D'Atri, and L. Tarantino. The Design of KIVIEW: An Object-Oriented Browser. In *Proceedings of the Second International Conference on Expert Database Systems*, April, 1988.

[NUWC97] S. Nestorov, J. Ullman, J. Wiener, and S. Chawathe. Representative Objects: Concise Representations of Semistructured Hierarchical Data. In *Proceedings of the Thirteenth International Conference on Data Engineering*, Birmingham, U.K., April, 1997.

[PGW95] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object Exchange Across Heterogeneous Information Sources. In *Proceedings of the Eleventh International Conference on Data Engineering*, pp. 251-260, Taipei, Taiwan, 1995.

[SK82] M. Stonebraker and J. Kalash. TIMBER - A Sophisticated Relational Browser. In *Proceedings of the Eighth International Conference on Very Large Data Bases*, Sept., 1982.

[Zlo77] M. Zloof. Query By Example. *IBM Systems Journal*, pp. 324-343, 16(4), 1977.