INTEGRATED QUERY AND SEARCH OF DATABASES, XML, AND THE WEB

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Roy Goldman
May 2000

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Jennifer Widom (Principal Advisor)

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Hector Garcia-Molina

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Jeffrey Ullman

Approved for the University Committee on Graduate Studies:

Abstract

The amount of information available on-line is proliferating at a tremendous rate. At one extreme, traditional database systems are managing large amounts of structured, well-understood data that can be queried via declarative languages such as SQL. At the other extreme, millions of unstructured Web pages are being collected and indexed by search engines for keyword-based search. Recently, XML— the eXtensible Markup Language— has emerged as a simple, practical way to model and exchange semistructured data across the Internet, without the rigid constraints of traditional database systems.

This thesis describes work towards unifying and integrating query techniques for traditional databases, search engines, and XML. First, we describe our contributions to the Lore DBMS for managing semistructured data, focusing on ways to enhance system usability for effective querying and searching. Next, we discuss algorithms and indexing techniques that enable effective keyword-based search over traditional and semistructured databases. We then describe how we have migrated and enhanced our research on semistructured data to support the subtle but important nuances of XML. Finally, we describe a new platform that enables efficient combined querying over structured traditional databases and existing Web search engines.

Acknowledgments

Above all, I thank my parents for all their love, help, support, and guidance.

I am grateful to my advisor Jennifer Widom for being so supportive and appreciative of my work and my ideas.

I would also like to thank the other members of the Stanford Database Group— especially Hector Garcia-Molina, Jason McHugh, Shiva Shivakumar, and Jeff Ullman— for helping me shape and refine my ideas.

I thank all the Lore developers for helping to build a platform for much of my research: Brian Babcock, Andre Bergholz, Vineet Gossain, Kevin Haas, Matt Jacobson, Jason McHugh, Svetlozar Nestorov, Dallan Quass, Anand Rajaraman, Hugo Rivero, Michael Rys, Raymond Wong, Beverly Yang, and Takeshi Yokokawa.

Finally, I thank my co-authors: Serge Abiteboul, Sudarshan Chawathe, Arturo Crespo, Hector Garcia-Molina, Kevin Haas, Qingshan Luo, Jason McHugh, Svetlozar Nestorov, Dallan Quass, Anand Rajaraman, Hugo Rivero, Narayanan Shivakumar, Jeff Ullman, Vasilis Vassalos, Suresh Venkatasubramanian, Jennifer Widom, Janet Wiener and Yue Zhuge.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

With the advent of the World-Wide Web we have seen enormous growth in the amount of information available on-line. From data published directly from traditional, well-structured databases, to the huge number of unstructured hand-written HTML pages, to the increasing amount of *semistructured* data [Abi97], the Web brings it all together into one giant amalgamation of information.

We can classify online data along a spectrum of how much structure exists in (or is imposed on) the data. At one extreme, traditional relational, object-oriented, and object-relational databases store large amounts of rigidly structured, typed data. Query languages such as SQL and OQL [Cat94] exploit the rigid structure imposed on the data to enable expressive, declarative queries. At the other extreme, many HTML pages exhibit little discernible structure, since they are manually-created documents. In general, it is extremely difficult if not impossible to capture the data relationships within unstructured HTML documents through a traditional database system. Instead, from the field of information retrieval (IR), effective technologies are available to support keyword-based searches over document data. Such searches are inherently less precise than SQL queries; as a consequence, results are often ranked, requiring interaction from a user to focus in on relevant data. Surprisingly, despite their similar goals, there has been very little research overlap between the fields of IR and databases.

To help bridge these two extremes, research and industry have focused recently on semistructured data— data that does exhibit some structure but is either too irregular or changes too often to be constrained by the table schemas and/or object classes required by traditional DBMSs. Semistructured data models, query languages, and database systems have converged around a graph-based data representation that combines data and structure into one simple data format [Abi97]. XML [XML98], which has emerged as a new standard for information interchange across the Web, can

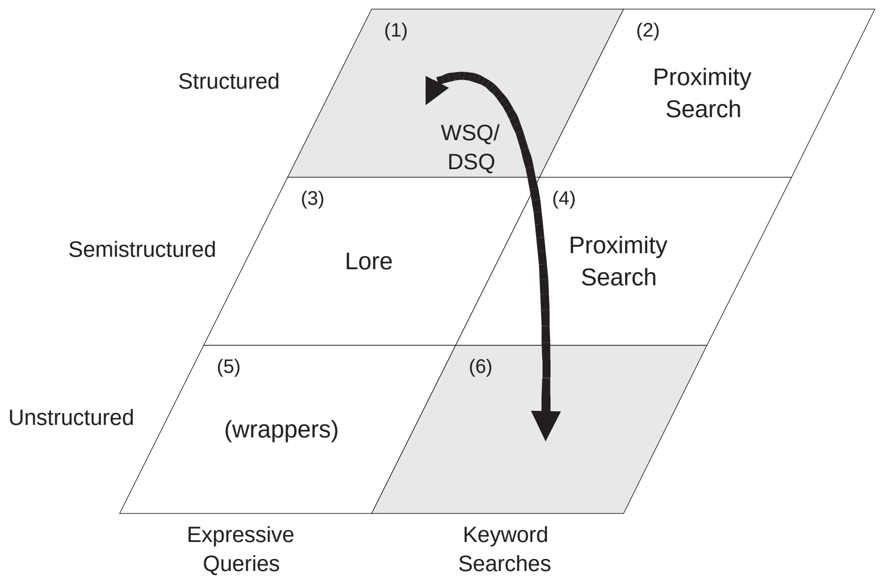


Figure 1.1: Query Functionality Matrix

be seen as one such semistructured data model. Among many other interesting research topics, an important issue is how to query and search a semistructured database effectively. Intuitively, expressive queries, keyword searches, or some combination of the two could be applied.

1.1 Matrix of Data Representations and Query Techniques

Let us view this spectrum of data representations and query techniques as a matrix, shown in Figure 1.1. Along one dimension, we have structured, semistructured, and unstructured data. Along the other we have expressive declarative queries and simple keyword searches. Two entries in our matrix are already accounted for: structured databases have long supported expressive queries (Entry 1), and unstructured data has traditionally been searched using keyword-based IR techniques (Entry 6). As the Web continues to unify all types of data in one medium, it will be increasingly important for users to be able to query all data seamlessly with the same techniques. In this thesis, we describe work that fills in remaining matrix entries and helps tie them together as well.

Before exploring our contributions in detail, let us consider each of the six entries in our matrix with respect to a given scenario: querying or searching a collection of data about movies, including actors, producers, writers, etc.

1.1.1 Traditional Databases

Entry 1 represents traditional database functionality. We can imagine that the movie data is stored in a relational database, with tables for movies, actors, producers, writers and so on, along with additional tables (or foreign-key relationships) for relating these elements. Traditional query languages such as SQL enable expressive queries over this data. As one example, it is a simple matter to write a query to " Find all movies made since 1980 that have grossed more than $100,000 million." Note that an object-oriented database system also corresponds to this matrix entry, where instead of tables we would have classes, and we would use a language such as OQL instead of SQL. Despite many proven advantages of database systems, traditional DBMSs have some deficiencies. First, we must be quite certain ahead of time of exactly what kind of data we wish to store. Data must be entered carefully, conforming to the specific structure laid out ahead of time. If we decide over time to store additional (or different) data in the database, traditional DBMSs are not amenable to rapidly evolving database structure. Similarly, data demonstrating many irregularities can also cause problems for traditional database systems; e.g., for relational systems, NULLs can proliferate and cause counterintuitive results. Finally, as powerful as SQL is, it is not an easy language for casual users. While interactive forms can make it easy for users to complete predefined parameterized queries, SQL has no inherent support for processing keyword searches over an entire database. Yet the Web has shown that keyword expressions are an important tool for interactive query sessions.

1.1.2 Traditional IR Systems and Search Engines

At the opposite corner of the matrix, Entry 6 represents traditional information retrieval systems and search engines: keyword search expressions over a collection of documents. In our movie scenario, we could imagine a hand-written (or machine generated) document repository with one document per movie, along with perhaps separate documents for the more famous actors, directors, and producers. This environment places few restrictions on the designers of the data collection: since all information is simply in document form, designers are free to add (or omit) any information for any film or film industry worker. Designers have complete flexibility to set or change the format of each document as well. In IR systems, searches are based on the presence (or lack of presence) of keywords. As a simple example, a user could quickly find all documents containing the string " Star Wars" . Results are often ranked based on the system's determination of relevance. For example, a document containing " Star Wars" many times might be ranked ahead of a document containing it only once. As a more complicated example, we could perform a *Boolean* search to see if Harrison

Ford and Carrie Fisher starred together in any movies besides the Star Wars series. The search would be expressed as: "Harrison Ford" and "Carrie Fisher" and NOT "Star Wars" and NOT "Empire Strikes Back" and NOT "Return of the Jedi".

1.1.3 Expressive Queries Over Unstructured Data

Considering matrix Entry 5, we note that in general it is inherently difficult or impossible to support expressive queries over unstructured data sources. As explained in Section 1.1.2, in a collection of unstructured documents about movies, there are no restrictions on structure, and the document designer has no tools for explicitly designating the semantics of the document's textual content. Our example query in Section 1.1.1 to find the high-grossing movies of the 80s and 90s would be impossible to execute directly over the documents (without making assumptions about document structure), since it requires some semantic understanding of the numbers and text appearing in each document.

When structural patterns do exist in documents, *wrappers* can be used over the unstructured data to expose the documents as richer structures [AK97, CDSS98, HGMC$^+$97, PGGMU95, RS97]. This approach is inherently brittle since it must be based on assumptions about the composition of documents that may have been generated without any guaranteed constraints. Further, it may be difficult to mold a large collection of (potentially varied) documents into the rigid data models of traditional DBMSs.

1.1.4 Semistructured Data

As a result of the inherent problems discussed in Section 1.1.3, over the last few years research and industry have developed semistructured data models, languages, and systems. XML is a prime example of a semistructured data model. XML supports a hierarchical model that allows data and structure to be mixed in one simple format, without the rigid constraints of a relational or object-oriented schema. While HTML uses *tags* to specify presentation, XML uses tags to indicate meaning. In our movie scenario, document designers could explicitly assign structural tags to the different document elements to enable expressive queries. For example, the year in which a movie was created could be bracketed by <YEAR> and </YEAR> tags. Similarly, we could designate the gross revenue for each movie within a document along with basic information such as the movie title. At a high level, semistructured data adds enough structure to enable expressive queries, while still avoiding the rigidity of traditional DBMSs.

To enable users to manage and query semistructured data, we have built the *Lore* database management system [MAG+97]. The primary query interface to Lore is *Lorel*, an OQL-like language for issuing declarative queries over a semistructured database [AQM+97]. In our matrix, Lore addresses Entry 3. There are many interesting research issues associated with managing and querying semistructured data. One issue we will focus on in particular is how we handle the fact that semistructured databases do not include an explicit, predetermined schema. Without some information about the tags in a database and their nesting pattern, it may be difficult to formulate meaningful queries.

1.1.5 Keyword Search Over Semistructured And Structured Databases

Keyword-based search is very useful for unstructured documents, and often is the only way to query such data. Keyword search also can be very useful over more structured data, since it is inherently simple for users to master and often is sufficient for the task at hand. However, some IR concepts and algorithms must be reconsidered in a database setting. In particular, *proximity search* benefits from a new approach in a database setting. Traditionally, proximity search in IR systems is implemented using the "near" operator. If we search our document collection for "Harrison Ford" near "Carrie Fisher", we are looking for documents where those two names appear "close" to each other, where closeness is measured by textual proximity. In this sense, proximity search is a relatively simple, "intra-object" operation: we measure proximity along a single dimension (text) in each document. Now, suppose that we have fully migrated our movie document collection to XML. Each movie might begin with a <MOVIE> tag, followed by nested tags for that movie's actors, producers, etc. In this setting, we want to account for "structural proximity" in the database, while textual proximity may not be relevant. For example, if Harrison Ford and Carrie Fisher both star in the same movie, then they will both be subelements of a specific <MOVIE> element. In the textual representation, however, there may be many other actors lexically in between these actors. Similarly, we may find that the last actor listed for some movie X is textually close to the first actor listed for an adjacent movie Y— but this doesn't mean that the two actors are related in any way. Thus, we need to extend the notion of proximity search to handle the structure inherent in a semistructured database. This work fills in Entry 4 in our matrix.

As we will see in Chapter 5, algorithms and techniques for performing proximity search over a graph-structured (semistructured) database are applicable to a traditional relational or object-oriented database as well. We can (logically) translate a relational database into a graph based on the schema and on primary/foreign key relationships. (Details are given in Chapter 5.) We can then use our proximity search techniques to measure the distance between database elements based on the graph representation. Viewing an object-oriented database as a graph is of course even simpler. By combining proximity search with traditional indexing techniques for identifying tables or attribute values that contain given keywords, we can provide keyword-based search (and browsing) for traditional databases. Thus, our work on proximity search applies to matrix Entry 2 as well.

1.1.6 Combining Database Queries and Keyword Search

Beyond filling out the different matrix entries, integrating query and search over the Web requires strong interoperation between different types of systems. One case we focus on is the integration of Entries 1 and 6. Traditional database systems for structured data are very different from (and incompatible with) IR-based Web search engines that index millions of unstructured (and unrelated) HTML documents for keyword-based searches. Yet there are many cases where it makes sense to query both kinds of systems at once. In our movie scenario, we may want to leverage the vast amount of information in Web pages to support expressive queries over a local structured database like the one described in Section 1.1.1. For example, we might want to rank all Star Wars actors by how often their names appear on the Web. We could use the structured database to accurately determine all of the Star Wars actors, and then we combine this data with results from a set of Web searches to compute our final result. Such functionality requires tight coupling between very different systems, and optimal execution requires that we take advantage of the fact that existing Web search engines are designed to handle many searches concurrently.

1.2 Research Contributions and Thesis Outline

1.2.1 Lore

In Chapter 2, we introduce *Lore* [MAG$^+$97], a database management system developed by many people from scratch at Stanford specifically to support semistructured data. Originally, Lore was designed solely to support a data model called *OEM*, for *Object Exchange Model*. In OEM, structure and data are combined into a simple, graph-based object model. Atomic data is stored in leaf objects, and relationships are indicated via textually labeled directed edges between objects. The primary query interface to Lore is Lorel [AQM$^+$97], a declarative query language based on OQL [Cat94]. Chapter 2 serves as a basis for later chapters and is primarily a summary of work done by the many contributors to the Lore project since 1995. The work in this thesis related to Lore focuses primarily on research issues related to a user's perspective of the system.

1.2.2 DataGuides

In traditional database management systems, the schema defining the structure of the data is fixed. For a given relational database, a fixed set of tables and their attributes guide all query construction. Similarly, declared classes guide queries in object-oriented DBMSs. But in a semistructured database, the schema is not declared ahead of time. Rather, structure and data are mixed into one simple data format. Further, the structure may be irregular and may change often and significantly over time. Without the guidance provided by a separate schema, a user may not be able to construct meaningful queries, and a query may become less useful if the structure of the data changes. To address these issues, in Chapter 3 we describe a novel database feature called the *DataGuide* [GW97]. A DataGuide is a concise, accurate structural summary of a semistructured database. The DataGuide is generated dynamically from the database, and is modified dynamically as the database structure evolves. In many ways, the DataGuide serves the role of " schema" in a semistructured database: it is a valuable tool for guiding query formulation, and it can be used for query optimization as well. The DataGuide also can be used within a graphical user-interface for interactive specification of Lorel queries. Note that the DataGuide never restricts the data— it always conforms to the structure in the database. Our discussion in Chapter 3 includes formal DataGuide definitions and gives algorithms for DataGuide construction and maintenance. Performance results for DataGuide construction are given. We also show how the DataGuide can be used as an index to improve query processing performance.

For certain databases, DataGuide construction can be prohibitively expensive. In these situations, we may want to relax our definition of a DataGuide for better performance. Hence, in Chapter 3 we also describe *Approximate DataGuides*, which can be smaller and require shorter construction times than exact DataGuides. Despite the relaxed definition, we show that approximate DataGuides are still useful in many scenarios.

1.2.3 Interactive Queries and Keyword Search for Semistructured Data

DataGuides make it easier for a user to explore structure and pose queries over a semistructured database. Users familiar with searching and exploring the Web are comfortable with an iterative process of searching, querying, and exploring. In Chapter 4, we formalize an approach and introduce associated techniques that enable a user to query and search a semistructured database iteratively, each time focusing in towards data of interest. In this chapter we explain how the system supports keyword-based searches, and we also discuss how to build DataGuides dynamically, in order to summarize query results and enable formulation of further refining queries.

1.2.4 Proximity Search

As described earlier, the IR notion of proximity search should take on a different meaning when applied to a database with some structure. While traditional proximity search is a relatively simple operation performed along a single dimension (text), applying proximity search to databases is a more difficult problem. In a semistructured database, for example, data is decomposed into many small nested data objects or elements. We can view such a database as a graph of data elements, with edges representing relationships between these elements. We can weight these edges as well, according to the "strength" of the semantic relationships— where a smaller weight on an edge indicates a stronger bond between the two elements it connects. In this setting, we can measure proximity as the shortest graph distance between data. Similarly, we can view any relational or object-oriented database as a graph of interrelated data elements as well. In a relational system, these elements might be attribute values, tuples, or tables, related by containment or by primary/foreign key constraints. In Chapter 5 we discuss proximity search in databases in detail. We develop a framework, algorithms, and a new indexing technique that supports proximity search over any semistructured or structured database. We then demonstrate how we have integrated proximity search into Lore to support interesting interactive searches over databases.

1.2.5 XML

XML is emerging as a new standard for data interchange across the Internet. XML is a text-oriented language that implies a data model very similar to the semistructured data models proposed by researchers, including the OEM model we use in Lore. However, there are differences between XML and OEM, many of which come from XML's heritage as a document language rather than a data model. For example, representing graph structure today in XML is hardly elegant. Perhaps the most striking difference is that an XML document has an inherent ordering among its elements. In contrast, OEM and most other semistructured data models assume that subobjects are always unordered. That is, an object can have a set of subobjects, not a list. In Chapter 6 we discuss the migration of the Lore system to an XML-based data model, focusing particularly on the impact of an ordered data model on DataGuides and proximity search.

1.2.6 WSQ/DSQ

Finally, in Chapter 7 we describe our work on integrating keyword-based search with structured database queries. We call this work WSQ/DSQ, pronounced "wisk-disk." WSQ/DSQ stands for Web-Supported (Database) Queries/Database-Supported (Web) Queries. From the WSQ angle, we use the results of Web searches to enhance and augment queries over traditional databases. From the DSQ angle, we can use known data relationships in a traditional database to help guide users as they search, and explain search results in terms of well-understood data. Our focus in this thesis is on WSQ. We show how we can model any Web search engine through *virtual tables*— tables that "look" like normal tables to a query processor but whose tuples are actually dynamically computed rather than physically stored in the database. Through this abstraction, reading tuples from our virtual tables corresponds to issuing Web searches. Further, we can use selection and join conditions on our virtual tables to parameterize the searches. This approach provides a powerful mechanism for integrating Web search results with queries over a relational DBMS. Unfortunately, the latency for each call to a Web search engine is extremely high. A traditional database query processor will sit idle during each call, resulting in extremely inefficient WSQ queries. Since existing Web search engines can process many concurrent requests effectively, the traditional approach is not making good use of available resources. Hence, we introduce a new query processing technique called *asynchronous iteration* that allows a conventional query processor to issue many concurrent searches with low overhead. This technique is a general one, applicable to many information integration scenarios.

1.3 Related Work

In this section we discuss some work that is broadly related to the main contributions of this thesis. Specific, more detailed discussions of related work are provided in each chapter.

Overviews of traditional database systems and query languages can be found in many sources (e.g., [UW97], [GMUW00]), while [Sal89] provides a thorough overview of the field of information retrieval, the technology many modern Web-based search engines are based on. Proximity search in IR systems is discussed in [Sal89], and to our knowledge we were the first to introduce IR-style proximity search into databases.

The field of semistructured data has seen a flurry of innovation over the last few years. Semistructured data models and query languages are discussed in [Abi97, Bun97, PGMW95]. Database systems for managing semistructured data are described in [MAG+97, FFLS97]. *XML* [XML98] has emerged as a standard format for interchanging semistructured data, and much of the original research on semistructured data has migrated to support XML [DFF+99a, GMW99]. In general, our work on DataGuides is complementary to the basic data models, query languages, and systems that have been designed for semistructured data; DataGuides are general enough to apply to any graph-based data model with unconstrained schema. DataGuides were inspired by initial work at Stanford on *representative objects* [NUWC97]. Using DataGuides as an interactive query tool is related to the seminal work on Query By Example [Zlo77]. DataGuides have since been cited as springboards for improved path indexing [MS99] and XML compression [LS00].

Our work on integrating Web search engines with traditional database systems is related in overall motivation to a large body of work on information integration, including the creation and optimization of query plans over external sources with varying query processing capabilities, e.g., [HKWY97, LRO96, Mor88, RSU95, YLGMU99]. Our specific work of integrating document repositories with a structured database is superficially related to work in [CDY95] and [DM97], though as we discuss in Chapter 7, our work turns out to be quite different since it is focused specifically on using existing Web search engines.

Chapter 2

The Lore DBMS for Semistructured Data and XML

This chapter provides an overview of *Lore*, a database management system designed specifically for managing *semistructured data*, such as data encoded in XML. Lore was built from scratch at Stanford University over the past five years, and it is a complete prototype multi-user database system. As a research platform, Lore has enabled much of our specific work on DataGuides and proximity search. This chapter serves to provide a setting for this work— describing the Lore data model, the *Lorel* query language, and the overall system architecture.

Lore was conceived to address several drawbacks of traditional relational or object-oriented database systems. These systems require all data to adhere to an explicitly specified, rigid schema. In some scenarios, this constraint may be too restrictive: data for a desired application may be irregular and thus not conform to a rigid schema. Further, it may be difficult to decide in advance on a single, correct schema. The structure of the data may evolve rapidly, data elements may change types, or data not conforming to the previous structure may be added. In traditional database systems, modifying the schema may have huge implications— often requiring data to be migrated wholly to the new structure, and requiring application code to be rewritten. We call data with irregular or fast-changing structure *semistructured data*— see [Abi97] for further discussion.

The semistructured data managed by Lore is not constrained by a schema, and the data may be irregular or incomplete. In general, Lore attempts to take advantage of structure where it exists, but Lore also handles irregular data as gracefully as possible.

Lore was designed to store and query data natively according to a simple object model called the *Object Exchange Model (OEM)*, introduced originally in the *Tsimmis* project at Stanford [PGMW95].

In 1998, as *XML* (the *eXtensible Markup Language*) emerged, we noticed an obvious similarly between XML's nested, tagged structure and OEM. In Chapter 6, we will discuss how we migrated Lore to support XML; until that chapter, all of our discussions are in terms of OEM, a somewhat simpler model. Regardless, there is a straightforward two-way conversion between OEM and XML— hence, all of our contributions to improve the management of OEM data are applicable to the management of XML as well.

Lorel, for Lore language, is the declarative query language that we designed for Lore. Lorel is as an extension of *OQL*, the *Object Query Language* [Cat94, BDK92]. Lorel augments OQL with extensive type coercion and powerful path expressions for effectively querying semistructured data. OEM and Lorel are reviewed briefly in this chapter; for details see [AQM+97].

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

First, query processing introduces a number of challenges. One obvious difficulty is the absence of a schema to guide the query processor. In addition, Lorel includes a powerful form of navigation based on path expressions, which requires the use of automata and graph traversal techniques inside the database engine. The indexing of semistructured data and its use in query optimization is an interesting issue, particularly in the context of the automatic type coercion provided by Lorel. While some of our DataGuide will work touch on query processing and optimization, these topics are covered in more detail in [MW99c, MW99b, MW99a, MWA+98].

With respect to Lore, the focus of the work in this thesis has been on several fronts:

- *DataGuides* replace traditional database schemas in Lore. A DataGuide enables users to browse the structure of a Lore database, it can be very helpful for formulating meaningful queries, and it is used for several internal features of the database system. DataGuides are described in detail in Chapter 3.

- We have integrated our work on *proximity search* in databases into Lore, supporting effective keyword-based search of Lore databases. We describe our model for sessions of interactive queries and keyword-based searches in Chapter 4. Proximity search is described in detail in Chapter 5.

- We designed an Application Programming Interface (API) that makes it easy for any client program to connect to and interact with a Lore database.

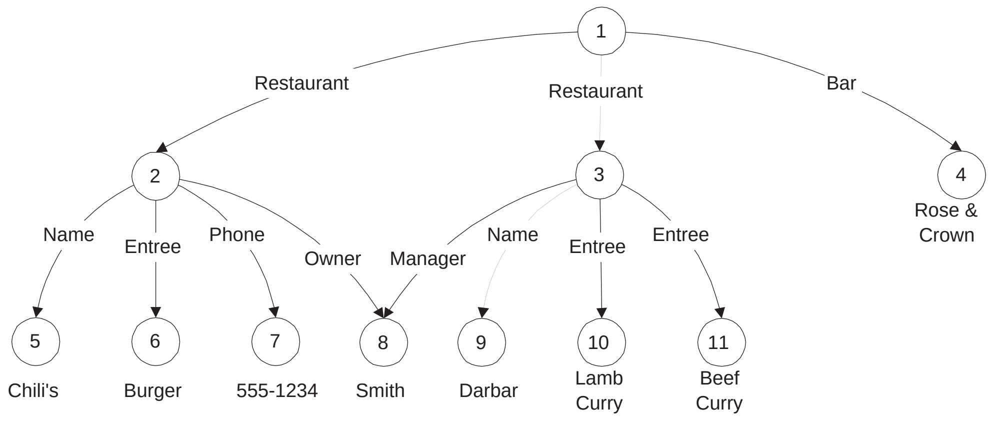


Figure 2.1: A sample OEM database

To showcase these features, we also developed a Web-based user interface to the Lore system via our API. For several years, this interface has enabled online users to search, query and browse Lore databases interactively. Screenshots from our user interface appear in the chapters on DataGuides and proximity search.

In the rest of this chapter, we summarize the Lore data model, the Lorel query language, and Lore's overall architecture. We end the chapter with a discussion of work related to Lore.

2.1 Object Exchange Model (OEM)

In its original design, Lore managed only OEM data. OEM is a very simple data model. In essence, an OEM database can be thought of as a directed, labeled graph. In OEM, each object contains an object identifer (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 cycles are allowed. For more details on OEM and its motivation see [AQM$^+$97, PGMW95].

Figure 2.1 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. 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.

In many ways, OEM is a "common denominator" among data models, since almost any data model can be encoded in a graph-based representation. For example, mappings from other data models into a graph-based model are used to show the wide applicability of our work in proximity search (Chapter 5). Chapter 6 describes the differences between OEM and its more popular relative, XML.

2.2 Lorel Query Language

Lorel is the declarative language used to query and update Lore databases. Lorel is based on OQL, with extensions for rich path expressions, and for extensive coercion— both between atomic types and between singletons and sets. We present several demonstrative examples of Lorel. As an extremely simple one:

```
Select DB.Restaurant.Entree
```

returns in Figure 2.1 all entrees served by any restaurant, the set of objects {6, 10, 11}. (Lorel assumes that every database has one or more incoming labels that we refer to throughout the thesis as *entry points*; in this case we assume DB is an entry point to object 1.)

As another simple example, we may request the names of all restaurants that serve burgers:

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

In our example database, the answer to this query is {5}. Notice that the above queries do not have From clauses— in these cases, we assume that all paths are examined from the DB entry point. Further, common prefixes repeated in the query (e.g., DB.Restaurant) are required to match to identical paths during query processing. Alternatively, we could rewrite the above query as:

```
Select R.Name
From DB.Restaurant R
Where R.Entree = "Burger"
```

In this case, the From clause iterates over all DB.Restaurant objects, assigning each to R in turn for use by the Where and Select clauses.

To help deal with the semistructured nature of the data, Lorel supports "wildcards" within queries that match to labels during query processing. For example, the following query finds all Restaurant objects that have an atomic subobject that contains "Smith" (independent of the incoming label).

```
Select DB.Restaurant.Name
Where DB.Restaurant.% grep "Smith"
```

In our example database, $\{5, 9\}$ is the result to this query.

To further assist in querying semistructured data, Lorel supports regular expression operators within path expressions:

```
Select DB.Restaurant(.%)*.(Manager | Owner)
```

In this query, we are looking for Manager or Owner objects reachable along any path of 0 or more labels following any DB.Restaurant. If we replace the * with a + in the above query, we require at least one label to be traversed between the Restaurant and the Manager or Owner objects. In Lorel, (.%)* can be abbreviated syntatically using the special symbol #.

Lorel is a rich language, including an expression calculator, subqueries, existential and universal quantification, aggregation, and updates. We have given only a few very simple examples here to set the stage for our reported results. See [AQM+97] for a full presentation of Lorel.

2.3 System Architecture

The basic architecture of the Lore system is depicted in Figure 2.2.

Access to Lore is through a variety of applications or directly via the Lore Application Program Interface (API) mentioned earlier.

The Query Compilation layer consists of the parser, preprocessor, query plan generator, and query optimizer. The parser accepts a textual representation of a query, transforms it into a parse tree, and then passes the parse tree to the preprocessor. The preprocessor handles the transformation of the Lorel query into an OQL-like query. A logical query plan is generated from the transformed query and then passed to the query optimizer. The query optimizer uses statistics about the database and knowledge about what types of indexes are available to select an efficient physical plan for the query. The physical plan is then sent to the Data Engine layer.

The Data Engine layer houses the OEM object manager, physical query operators, tools for

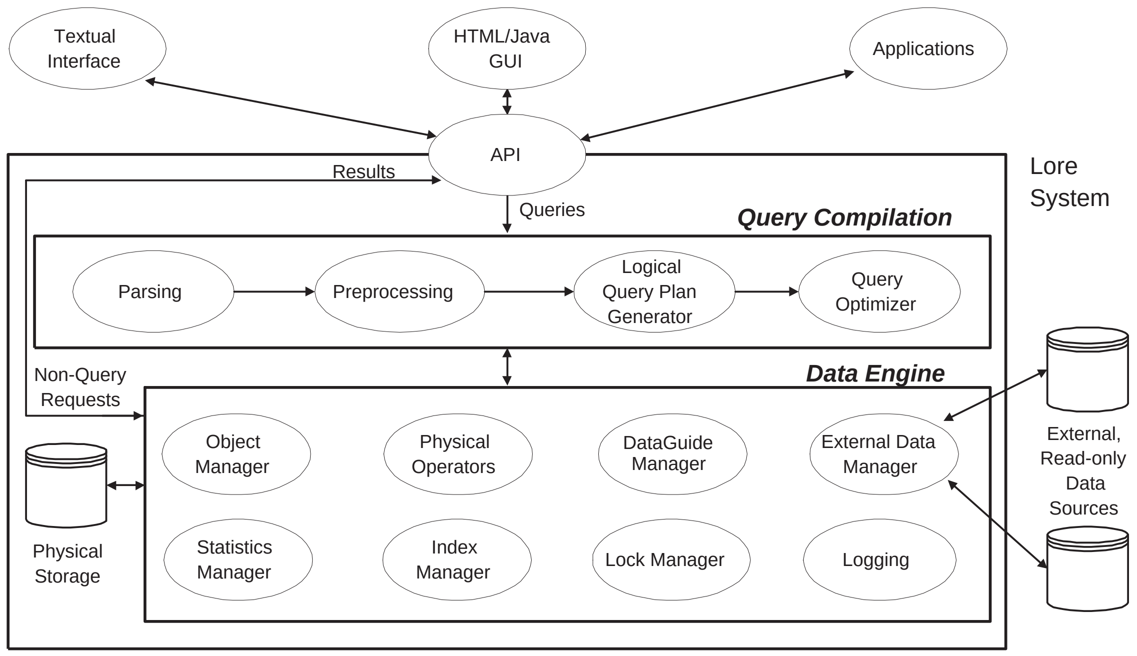


Figure 2.2: Lore architecture

concurrency control and recovery, and various utilities such as a statistics manager for query optimization and an external data manager for integration of external semistructured data. The query operators execute the generated query plans. The object manager functions as the translation layer between OEM and the low-level file constructs. It supports basic primitives such as fetching an object, comparing two objects, performing simple coercion, and iterating over the subobjects of a complex object. In addition, some performance features, such as a cache of frequently accessed objects, are implemented in this component.

The index manager creates and maintains several different kinds of Lore indexes, including the indexes for keyword and proximity search. The DataGuide manager is responsible for building and maintaining DataGuides and is discussed in detail in Chapter 3.

Lore is a product of many months of work by many people. Within the context of Figure 2.2, our specific contributions have been related to DataGuides, the HTML/Java user interface, the Lore API, and specialized indexes for keyword-based search.

2.4 Related Work

The Lore project started as a component within Stanford's *Tsimmis* project [PGMW95, PGGMU95, PAGM96, PGMU96], which studied the integration of heterogeneous data sources. Originally, Lore was intended to be used as a cache of OEM data during query processing in Tsimmis. Lore quickly evolved into its own research project as we became more interested in the many facets of managing semistructured data.

There have been several other research projects studying the management of semistructured database. At the University of Pennsylvania, researchers proposed *UnQL* [BDHS96], a powerful query language for semistructured data, based on a data model similar to OEM. While no prototype system using UnQL has been made available, the language has influenced the design of Lorel over time. At AT&T, researchers have developed *Strudel*, a system designed to simplify Web-site management and Web-page generation that includes its own *StruQL* query language [FFLS97, FFK$^+$99, FLS98]. Some of the Strudel team later contributed to the query language and data model for *XML-QL* [DFF$^+$99b], perhaps the first work on semistructured data done entirely in the context of XML. As XML has gathered more and more attention worldwide, several commercial products have become available for managing XML data. These products are described in Chapter 6, which discusses Lore's migration to support XML.

Chapter 3

DataGuides

In a traditional relational or object-oriented database management system, the data is formally separated from the specification of its *schema*, or structure. For example, in a relational database about movies, the database schema would specify several tables (e.g., Movies, Actors, Directors, Writers) and their attributes (e.g., the attributes of the Actors table might include Name, Birthdate, and Birthplace). The database schema must be specified completely before any data can be loaded into the system. A schema serves two important purposes:

- A schema, in the form of either tables and their attributes (for relational systems) or class hierarchies (for object-oriented systems), enables users to enforce and understand the structure of the database, and to form meaningful queries over it.

- The DBMS 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 browse a small database manually, 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.

As introduced in Chapter 2, semistructured data does not have an explicitly declared schema. To address the challenges faced by users and systems when a schema is not present, we introduce *DataGuides*: dynamically generated and maintained structural summaries of semistructured databases. Our work is cast in the context of the Lore system (Chapter 2). However, our contributions are applicable to any graph-based model for semistructured data, including XML.

This chapter 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,* which are well-suited for implementation and exploitation within a DBMS for semistructured data.
- We provide 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 integrated successfully into Lore. DataGuides are vital to Lore's user interface: users depend on the DataGuide to learn about the structure of a database so they can formulate meaningful queries. In addition, users may specify and submit queries directly from the DataGuide.
- We explain how a query processor can use a strong DataGuide to optimize query execution significantly, focusing on using a DataGuide as a *path index.*
- Because fully accurate DataGuides can be expensive to compute and store in the worst case, we describe *Approximate DataGuides*, which relax part of the DataGuide definition. Approximate DataGuides can be significantly cheaper to compute and store, yet they still are useful in many situations.

3.1 Chapter Outline

Section 3.2 provides the motivation and formal definition for DataGuides, along with an algorithm for constructing a DataGuide from a database. In Section 3.3 we present experimental results showing the time and space required to build and store typical DataGuides. Section 3.4 presents an incremental algorithm for DataGuide maintenance in response to database modifications. Section 3.5 describes how DataGuides are used in practice to browse structure and guide query formulation through a graphical interface to the Lore system. In Section 3.6 we see how a strong DataGuide can improve query processing in Lore, especially when used as a path index. In Section 3.7 we explain how we can relax the DataGuide definition to generate “ approximate” DataGuides that are still useful in many scenarios, but are less expensive to compute than fully accurate DataGuides. Finally, we conclude the chapter with a short discussion of related work and the impact of DataGuides on the field of semistructured data. In Chapter 4 we will discuss how we can use DataGuides to enhance

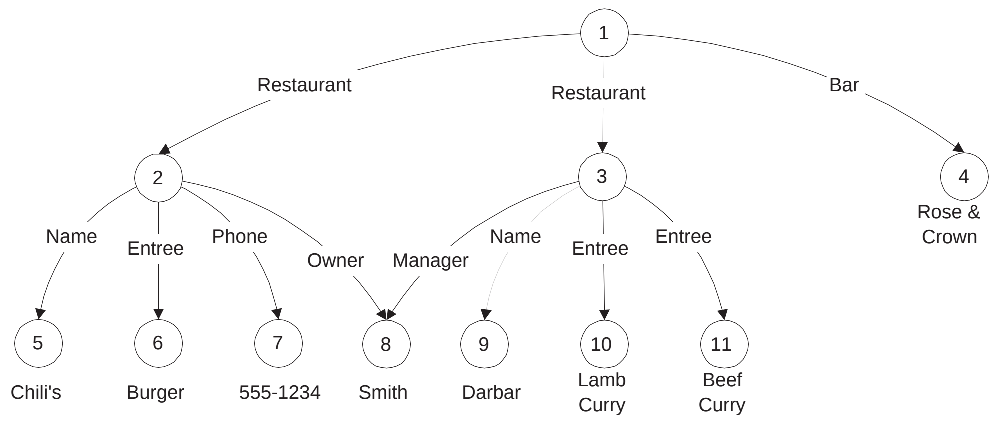


Figure 3.1: A sample OEM database (same as Figure 2.1)

interactive query and search sessions, and in Chapter 6 we will revisit DataGuides in the context of XML. Some of the work reported in this chapter was published initially in [GW97] and [GW99].

3.2 Foundations

In this section we motivate and define DataGuides and their properties, and we provide an algorithm for building them. We use the OEM data model as described in Chapter 2. DataGuides for XML are discussed in Chapter 6.

3.2.1 DataGuides

Consider Figure 3.1, the sample OEM database we introduced in Chapter 2. It serves as a basis for multiple examples throughout this chapter.

We now give several definitions useful for describing an OEM database and subsequently for defining DataGuides.

Definition 3.2.1 (**label path**) A *label path* of an OEM object o is a sequence of one or more dot-separated labels, $l_1.l_2.\cdots.l_n$, such that we can traverse a path of n edges ($e_1 \ldots e_n$) from o where edge e_i has label l_i. □

In Figure 3.1, Restaurant.Name and Bar are both label paths of object 1. In Lorel (Chapter 2), queries are based on label paths. For example, in Figure 3.1, a Lorel query might request the values of all Restaurant.Entree objects that satisfy a given condition. (Without loss of generality, we ignore for now the fact that Lorel label paths require an incoming label as indicated in Section 2.2.)

Definition 3.2.2 (**data path**) A *data path* of an OEM object denoted by oid o is a dot-separated alternating sequence of labels and oids of the form $l_1.o_1.l_2.o_2.\cdots.l_n.o_n$ such that we can traverse from o a path of n edges $(e_1 \ldots e_n)$ through n objects $(x_1 \ldots x_n)$ where edge e_i has label l_i and object x_i has oid o_i. □

In Figure 3.1, Restaurant.2.Name.5 and Bar.4 are data paths of object 1.

Definition 3.2.3 (**instance**) 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 3.1, Restaurant.2.Name.5 is an instance of Restaurant.Name and Bar.4 is an instance of Bar.

Definition 3.2.4 (**target set**) In an OEM object s, a *target set* of a label path l is a set t of oids such that $t = \{o \mid l_1.o_1.l_2.o_2.\cdots.l_n.o$ 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 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 3.1 is $\{6, 10, 11\}$. Note that two different label paths may share the same target set. Set $\{8\}$, for instance, is the target set of both Restaurant.Owner and Restaurant.Manager.

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 encodes 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 a 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.

Definition 3.2.5 (**DataGuide**) 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 3.2 shows a DataGuide for the source OEM database shown in Figure 3.1. Using a DataGuide, we can check whether a given label path of length n exists in the original database by

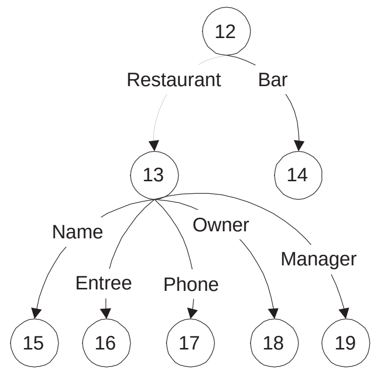


Figure 3.2: A DataGuide for Figure 3.1

considering at most n objects in the DataGuide. For example, in Figure 3.2 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 3.2, 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 3.2.4, 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], which proved that creating a DataGuide over a source database is equivalent to conversion of a nondeterministic 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 reported in Section 3.3 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, we know of no work that works to identify quickly those NFAs that may or may not require exponential time or space to be converted to equivalent DFAs.

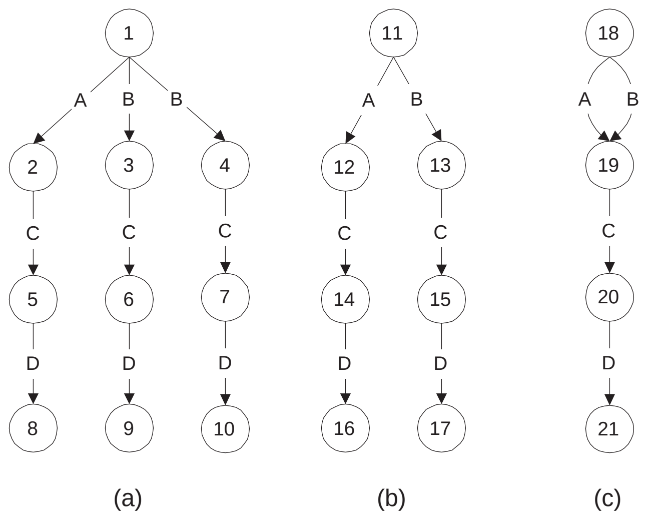


Figure 3.3: A source and two DataGuides

3.2.2 Existence of Multiple DataGuides

From automata theory, we know that a single NFA may have many equivalent DFAs [HU79]. Similarly, as shown in Figure 3.3, one OEM source database may have multiple DataGuides. Figures 3.3(b) and (c) are both DataGuides of the source in Figure 3.3(a). Each label path in the source appears exactly once in each DataGuide, and neither DataGuide introduces any label paths that do not exist in the source. Figure 3.3(c) is in fact *minimal*: the smallest possible DataGuide, in terms of total number of nodes. (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 (as suggested by [NUWC97]), furthering our goal of having as concise a summary as possible. 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 3.3(a), suppose we add a new child object to 10, via the label E. To correctly reflect this source insertion in Figure 3.3(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 3.3(c), we must do more work in order to somehow generate the same DataGuide as our updated version of Figure 3.3(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.

3.2.3 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 3.6. The following definition covers all of these examples.

Definition 3.2.6 (**annotation**) 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 3.3(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 3.3(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.

3.2.4 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 3.2.7 (**strong DataGuide**) 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 3.3(c) is not a strong DataGuide for Figure 3.3(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. Figure 3.3(b) is a strong DataGuide. Next, we prove that a strong DataGuide is sufficient for storage of annotations.

Theorem 3.2.1 *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. Therefore, $T_s(l) \neq T_s(m)$, since we know by Definition 3.2.6 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 3.4) and query processing (Section 3.6).

Theorem 3.2.2 *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

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 3.4: Algorithm to create a strong DataGuide

$t = u$, 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 3.2.1) 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. □

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 3.3(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 3.3(c) has only 4 objects, we cannot have a one-to-one correspondence.

3.2.5 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 3.2.2 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 complete algorithm is specified in Figure 3.4. 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.

3.3 Experimental Performance

As described in Section 3.2.1, computing a DataGuide for a source is equivalent to converting a nondeterministic 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 in the worst case require exponential running time and could feasibly generate a DataGuide exponentially larger than the source. Needless to say, we were 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.

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

Database	Source objects	Source links	Unique labels	Height	DataGuide objects	DataGuide links	DataGuide 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 3.1: DataGuide performance for operational Lore databases

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 were taken running the Lore system (Chapter 2) on a Sun Ultra 2 with 256MB RAM.

3.3.1 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. Our tree-structured database contains a snapshot of data imported from a large Web site covering many different sports (www.espn.com), 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 throughout the thesis.) 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. The version of the database used in the experiments in this chapter 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 3.1.

3.3.2 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 randomly the number of outgoing edges of any object from zero to the maximum fan-out (*irregular*).

For graph-structured synthetic 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 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 3.2. 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

DB No	Tree ?	Source Objects	Source Links	Hgt	Labs per Lvl	Fan-out	Full ?	Objs per Lvl	Bklink Freq/ Lvl	DG Objects	DG Links	DG 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 3.2: DataGuide performance for synthetic databases

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 in Sections 3.5 and 3.6. 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.

While it is impossible 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

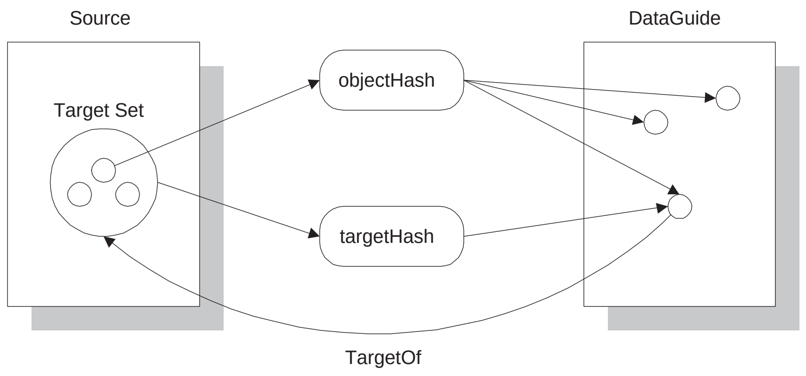


Figure 3.5: Data structures to support DataGuide maintenance

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— to the point of exhausting system resources during DataGuide construction. This problem is discussed in detail in Section 3.7, where we propose a relaxation of the DataGuide definition that provides better performance and yields a structural summary that is still useful in many situations.

3.4 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 3.4 for incremental maintenance. First, we list changes to the algorithm's data structures, as summarized in Figure 3.5.

- As we construct target sets in the DataGuide algorithm (in variables t1 and t2, Figure 3.4), 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 3.2.2). 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.

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, and we are not concerned with whether the edge represents an insertion or a deletion. (When adding an edge, v may or may not already exist in the database.) Note that the algorithm can handle the insertion of a complete subgraph directly, 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 3.4. 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 3.6. 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 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

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

HandleUpdate: algorithm to build a strong DataGuide over a source database
Input: U, a set of edge updates, each of the form $u.l.v$ (and global variables below)
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 3.6: Algorithm to update a strong DataGuide incrementally

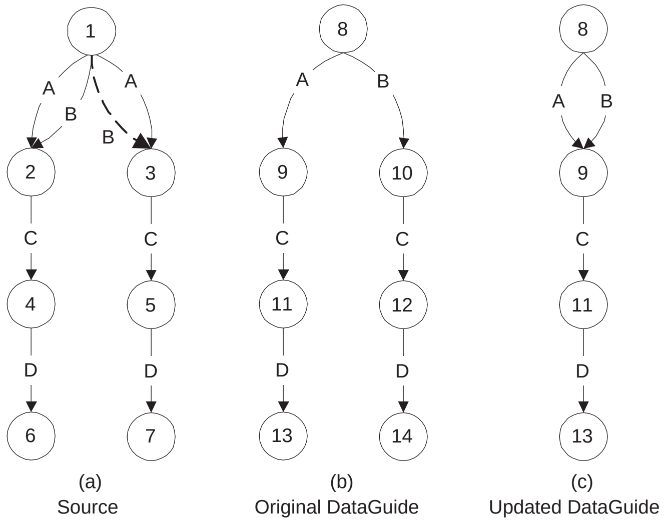


Figure 3.7: Insertion of an edge

hash tables.

Next, we trace two examples to demonstrate the algorithm.

EXAMPLE 3.4.1 Figure 3.7 shows one of the trickier cases for insertion. Figure 3.7(a), without the dashed B edge between objects 1 and 3, is our original source, and Figure 3.7(b) is a strong DataGuide for this source (with TargetOf links omitted). Suppose we insert the B edge. Handle-Update 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 3.7(c).

Notice that deleting the edge we just inserted would regenerate a DataGuide equivalent to Figure 3.7(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. □

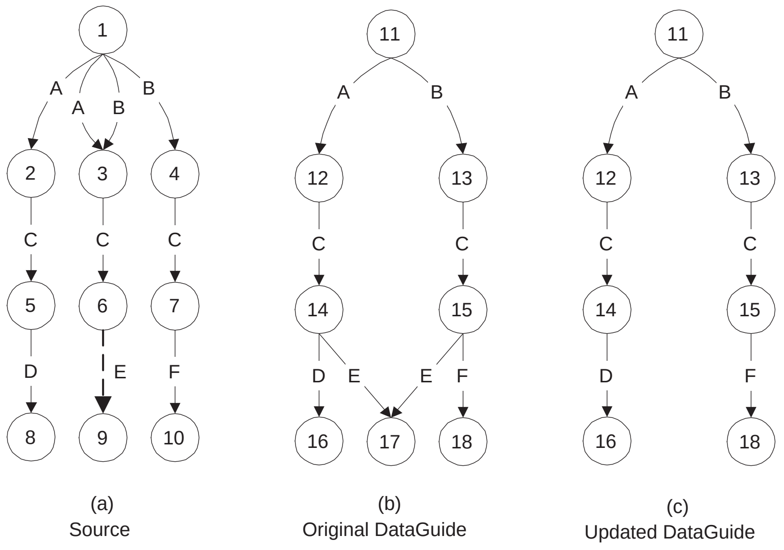


Figure 3.8: Deletion of an edge

EXAMPLE 3.4.2 We now demonstrate how the algorithm handles deletion in a case where we must recompute multiple sub-DataGuides. Figure 3.8(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 3.8(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 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 3.8(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.

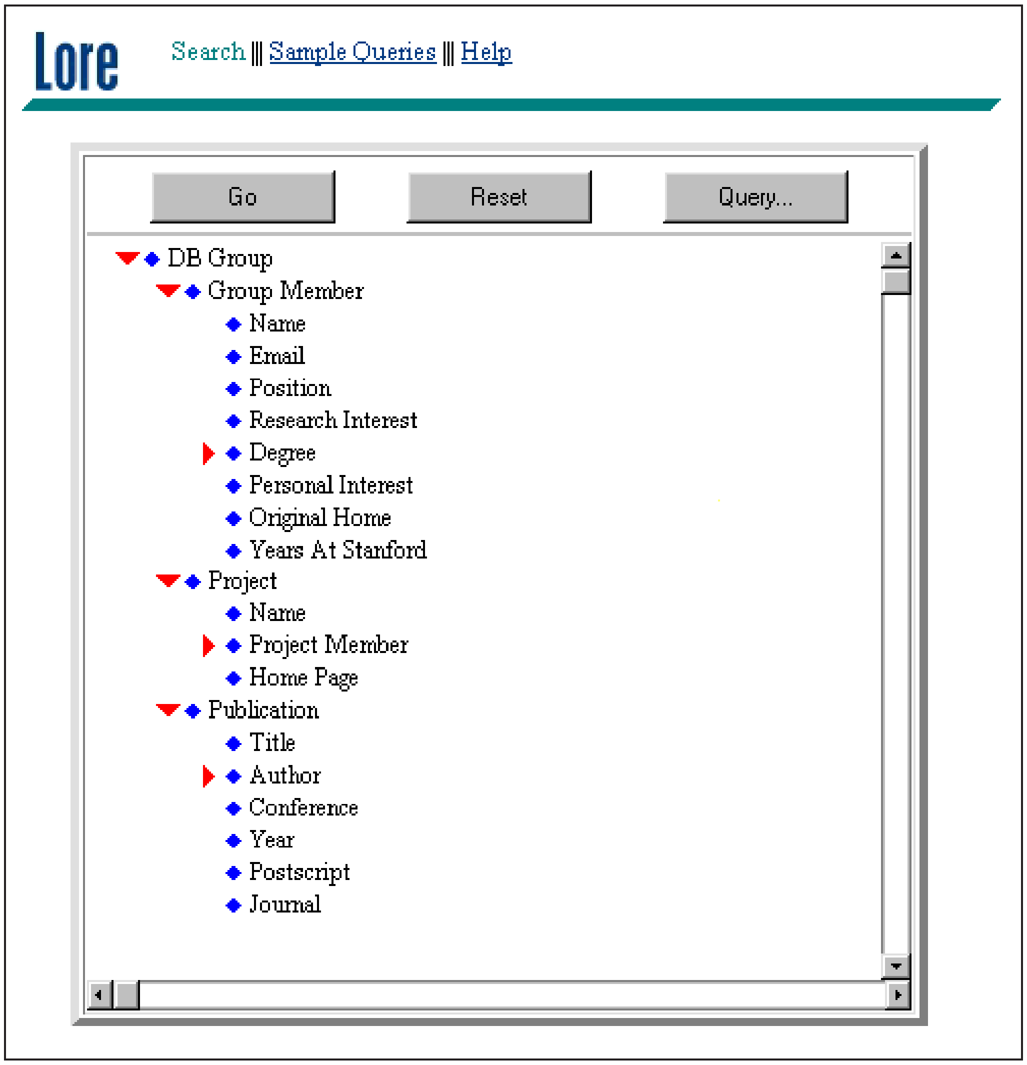
3.5 Using DataGuides for Query Formulation

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.

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 3.3. 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 3.9.

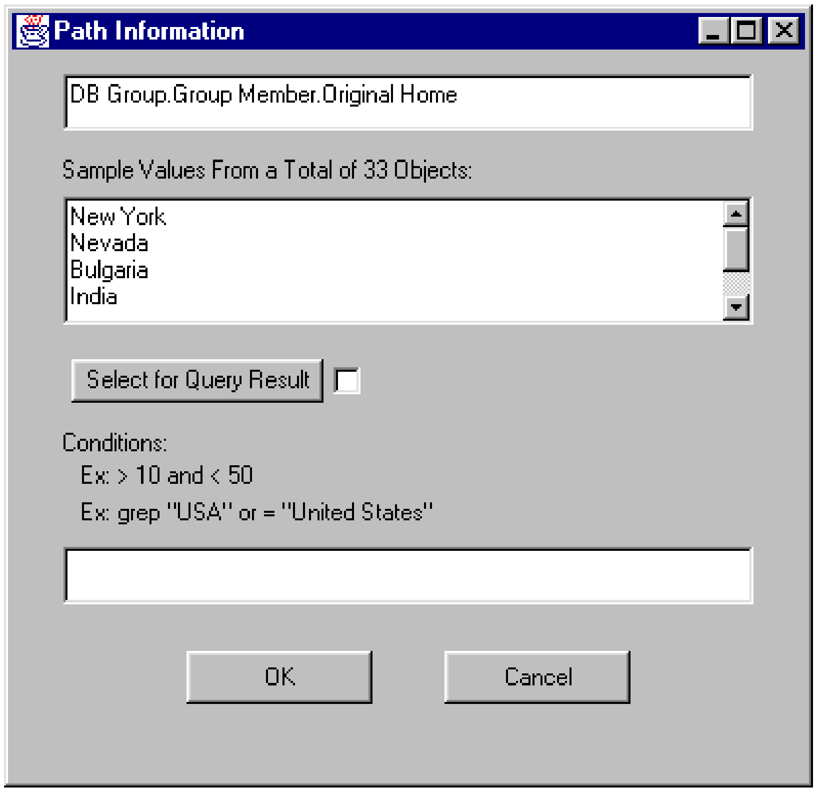
The user can explore the DataGuide by clicking on the arrows (triangles), which expand or collapse complex objects within the DataGuide. Immediately, we see how the DataGuide guides the specification of path expressions used in Lorel queries: every valid path expression must begin with the DB_Group label, 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 given 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+97] to write a query that attempts to encompass all possible formats, but in many cases a better approach is to examine sample values from the database. As described in Section 3.2.3, we can effectively store such sample values as annotations in the DataGuide. In Figure 3.9, notice that a diamond accompanies every label, corresponding to a distinct label path from the root. Clicking on the diamond brings up



DB Group
Group Member
Name
Email
Position
Research Interest
Degree
Personal Interest
Original Home
Years At Stanford
Project
Name
Project Member
Home Page
Publication
Title
Author
Conference
Year
Postscript
Journal

Figure 3.9: A Java DataGuide



DB Group.Group Member.Original Home

Sample Values From a Total of 33 Objects:

New York
Nevada
Bulgaria
India

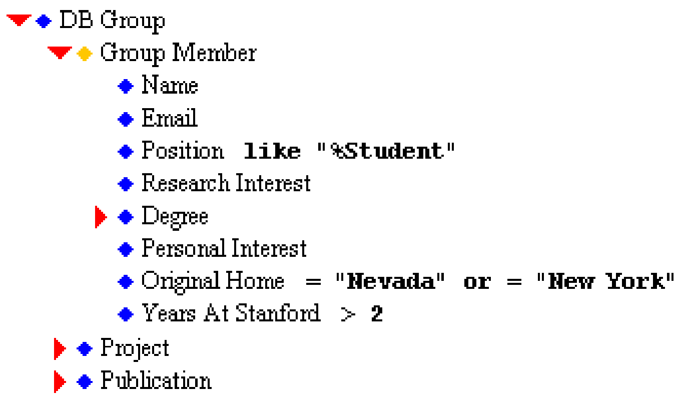
Conditions:
Ex: > 10 and < 50
Ex: grep "USA" or = "United States"

Figure 3.10: DataGuide path information

a dialog box such as the one shown in Figure 3.10, which was obtained by clicking on the diamond next to the Original Home label.

The top portion of the dialog box identifes 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. A fixed number of values are chosen arbitrarily from the database, although clearly there is room to 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 3.4.

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 specifed using common comparison and boolean operators, as well as custom operators such as the UNIX utility grep and the SQL function like. (These comparisons correspond to Lorel " where"



```
▼ ◆ DB Group
    ▼ ◆ Group Member
          ◆ Name
          ◆ Email
          ◆ Position  like "%Student"
          ◆ Research Interest
        ▶ ◆ Degree
          ◆ Personal Interest
          ◆ Original Home  = "Nevada" or = "New York"
          ◆ Years At Stanford  > 2
    ▶ ◆ Project
    ▶ ◆ Publication
```

Figure 3.11: A DataGuide query specification

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 3.11 shows the DataGuide after a user has specified to select all 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 3.11, 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.

DataGuide queries can specify any Lorel query with simple path expressions (no path wildcards, recall Section 2.2) and " where" clauses that are conjunctive with respect to unique path expressions. Also, all value comparisons must be made against constants. It would not be difficult to expand the expressive power of DataGuide queries; e.g., adding 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 3.11 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 sample values from 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. As one step in this direction, Chapter 4 describes how we can generate special DataGuides over query results to improve interactive query sessions.

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

3.6 DataGuides as Path Indexes

In this section we discuss how the information maintained by a strong DataGuide can be used to speed up query processing significantly 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 had not been addressed prior to our work. 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 3.4 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 locate all source objects reachable via that path, independent of the size of the source. (Of course, to examine all of these objects takes time proportional to the size of the target set.) In this analyze a sequence of sample query executions 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. In Lore, 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 3.6.1 We begin by tracing a very simple Lorel query over a sample database, showing how the DataGuide can reduce dramatically query execution cost. Suppose we wish to execute the following Lorel query (recall Chapter 2) over a database with structure similar to the Stanford Database Group database described in Section 3.5. It finds all Group_Member 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 is reachable along the path specified in the query. (Assume for now that the database contains 100,000 total Troff publications, even though only one is reachable along the path of interest.) 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. If instead we attempt to begin query processing by using a straightforward index to identify any object with an incoming Troff label, we will need to examine 100,000 objects.

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 3.4, the object in the DataGuide reachable by DBGroup.GroupMember.Publication.Troff includes as part of its value a TargetOf link to a special 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 works with any DataGuide (not necessarily a strong one) and was in fact suggested by [NUWC97].

□

EXAMPLE 3.6.2 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 can use a B-tree [Com79] 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, DB-Group.GroupMember.Publication.Year. Because the DataGuide algorithm in Figure 3.6 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 3.6.3 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 DB-Group.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 3.6.3 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 3.6.1, if either path does not exist then the query result is empty. Otherwise, we proceed as in Example 3.6.2 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 speed up common queries significantly. DataGuides have been integrated into Lore's cost-based query optimizer [MW99c]:

the optimizer automatically considers using a DataGuide path index as one of the potential strategies available for query evaluation. (Note that using a DataGuide path index is not necessarily always the best approach for evaluating a query, since we may need objects bound along a given path.)

The path index techniques we have described also apply to queries with more sophisticated path expressions. For example,

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

selects Publications either directly under DB_Group (since the ? makes the other labels optional) or under 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.

DataGuides also can be used to expand wildcards and regular expressions during query compilation. For example, consider the query:

```
Select DB_Group(.%)?.Publication
```

Recall from Section 2.2 that % matches any label, and the ? indicates that the label is optional. During query compilation, by consulting the DataGuide we could determine quickly all paths that satisfy the wildcard and regular expression in the current instance of the database. In this example, the query processor might use the DataGuide to conclude that DB_Group.Publication, DB_Group.Group_Member.Publication, and DB_Group.Project.Publication are the only paths that could satisfy the query. By expanding such wildcards and regular expressions at compile time, we guarantee that we will visit (at runtime) a subset of the objects that would have been visited with the original path expression, regardless of execution strategy. See [MW99a] for more details.

On a related note, Lore includes a *query warning system* that uses the DataGuide to warn users when a Lorel query includes path expressions that do not exist in the current instance of the database. One of the design goals of Lorel was to ensure that a given query can still return valid results even if some of the path expressions it refers to are not matched in the database. Hence, the warning system is a way to give a query writer feedback about invalid paths without halting query execution and returning an explicit error. For example, consider the query:

```
Select DB_Group.Project.#Hobby
```

If the query is run at a point when the only Hobby objects in the database are reachable via the path DB_Group.Group_Member.Hobby, then the query will return a warning that the specified path does not currently exist— a conclusion drawn by consulting the DataGuide.

3.7 Approximate DataGuides

As explained in Section 3.3, our algorithm for DataGuide construction performs well in many situations for both tree-structured and graph-structured databases. However, we have seen examples of highly cyclic databases that result in very poor performance. For example, we have a 4MB database for which DataGuide creation runs for several hours without terminating; we explain the cause of the problem in Section 3.7.1.

For many DataGuide uses, an “ approximate” summary of the database's structure can still be beneficial, yet much cheaper to compute. We define *Approximate DataGuides (ADGs)*, which relax certain aspects of the DataGuide definition. An ADG allows some inaccuracy yet retains properties that make it useful in numerous situations. This section presents two general approaches for building ADGs, describing algorithms and experimental results.

Recall our definition of a DataGuide (Definition 3.2.5), which has two requirements: (1) every label path in the source database exists exactly once in the DataGuide; (2) every label path in the DataGuide exists in the source database. Quite simply, an ADG drops the second requirement that all DataGuide paths must exist in the original database. Therefore an ADG may have “ false positives” but never “ false negatives” concerning the existence of database paths.

Let us reconsider five of the DataGuide uses described earlier in this chapter:

- *Query Formulation*: As described in Section 3.5, exploring an interactive DataGuide can be very useful for learning the structure of the database and formulating meaningful queries. A user exploring an ADG may see paths that do not actually exist in the database. If he formulates an unfiltered query over one of these “ false paths,” the query result may be empty, whereas the same query over an accurate DataGuide will always return at least one matching object.

- *Statistics*: As mentioned in Section 3.2.3, we can associate statistical information with database label paths by annotating DataGuide nodes with values. We can still associate statistics with every ADG object, hence we can store statistics for every rooted path in the database. However, some statistics may be based on a superset of the actual objects reachable along that path.

- *Path Index*: An index is expected to be exact, so using an ADG as a path index, as described in Section 3.6, is not feasible.

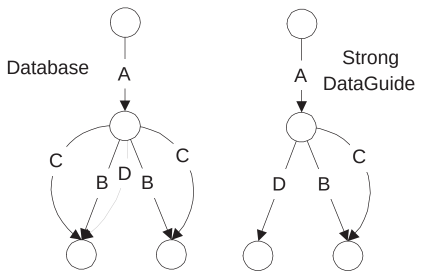


Figure 3.12: A sample OEM database and its strong DataGuide

- *Path Expressions*: In Section 3.6 we explained how we can use the DataGuide to expand wildcards and regular expressions at compilation time. Using an ADG, we may expand to a superset of valid path expressions. Such an expansion will not affect the correctness of query results, but it may degrade efficiency [MW99a].

- *Warnings*: Using the DataGuide for query warnings was also described in Section 3.6. The system may fail to warn the user that certain path expressions do not exist, but it will never incorrectly warn that a valid path does not exist.

We propose several strategies for building effective ADGs. In each case, we identify " similar" portions of the DataGuide and merge them. It is this merging process that may introduce superfluous paths. We see it as a requirement that all merging occur during construction— rather than as a post-processing step— because constructing a (regular) DataGuide is exactly the performance bottleneck we are trying to avoid. We discuss two general approaches to approximation:

- *Object Matching*: This approach is based on the hypothesis that two label paths in a database are " similar" if the sets of objects reachable via those paths are similar, i.e., they have a significant intersection.

- *Role Matching*: For this class of approximation, we decide whether two label paths are similar based on the paths themselves, without regard to the objects they reach.

3.7.1 Object Matching

Recall from Section 3.2.1 that the target set of a path is the set of all objects reachable via that path. In a strong DataGuide, each DataGuide object corresponds to the target set of all label paths that reach that DataGuide object. Two label paths in the DataGuide point to the same DataGuide

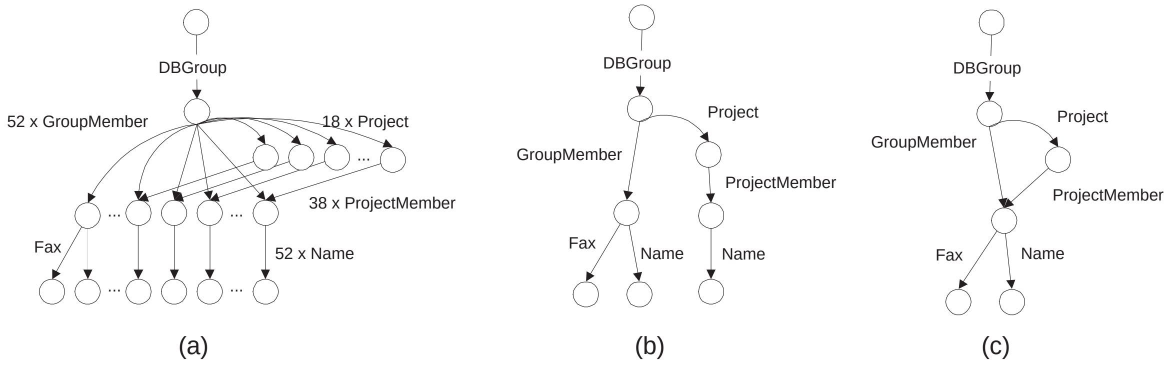


Figure 3.13: An OEM database, its strong DataGuide, and an approximate DataGuide

object if and only if the target sets of both label paths are exactly the same in the original database. Figure 3.12 shows a small database and its strong DataGuide. Notice that paths A.B and A.C in the DataGuide point to the same object because the target sets of A.B and A.C are the same in the database. The path A.D gets its own object in the DataGuide because its target set is different from the others.

The algorithm presented in Figure 3.4 creates a strong DataGuide by performing a depth-first exploration of the database, building up target sets of the label paths visited. Each target set is stored in a hash table. Each time the graph exploration generates a target set, the algorithm checks the hash table to see if that target set has already been discovered. If not, then a DataGuide object is created for that specific target set, the target set is added to the hash table (along with its corresponding DataGuide object), and the new object is linked into the DataGuide according to the path used to reach the target set. If, on the other hand, we have already seen the target set, then we can find its corresponding DataGuide object, and we add an edge in the DataGuide to that existing object.

Suppose that instead of requiring target sets to be exactly equal before equating their corresponding DataGuide objects, we instead allow DataGuide paths to point to the same object when their target sets are " almost" equal. In doing so, we may introduce DataGuide paths that do not exist in the database (false positives).

Consider Figure 3.13(a). Compare the target sets of DBGroup.GroupMember (52 objects) and DBGroup.Project.ProjectMember (38 objects). Because the target sets are not identical, each will correspond to a different strong DataGuide object. Note that all GroupMembers have a Name, and one who is not a ProjectMember also has a Fax. Figure 3.13(b) is the strong DataGuide for Figure 3.13(a).

Suppose that when comparing the target sets for the GroupMembers and ProjectMembers, we merged the corresponding DataGuide objects because of their significant intersection. We then build the "sub-DataGuide" over the union of the GroupMembers and the ProjectMembers. This case is shown in Figure 3.13(c). Performing the merge of DataGuide objects can introduce false paths: in our case the ADG incorrectly suggests that at least one ProjectMember has a Fax.

This object-matching approach to DataGuide approximation introduces many interesting issues:

- How do we define whether two sets are "similar"? One simple criterion (used in the remainder of this section) is to consider two sets X and Y similar when $|X \cap Y| / \max(|X|, |Y|)$ is above some threshold t.

- How does the DataGuide construction algorithm change? Again, we need to make our approximations during construction rather than reducing a constructed (full) DataGuide. This on-line approach unfortunately gives some importance to the way we traverse the original database to construct target sets. For example, we may decide that sets X and Y are similar enough to merge them into Z. At this point, the original sets disappear. Suppose we then encounter another set W. W may be similar to X, but not to the newly created set Z. If we would have traversed the database differently, W and X may have been merged. In addition, we may want to limit the number of times any given (original) target set can participate in a merge operation in an effort to bound the difference between a target set and the final object set it is a part of. Suppose that we have already constructed the "sub-DataGuide" for some target set X. If we encounter some target set Y that we decide is similar to X, there are two possible scenarios for the algorithm: if Y is a subset of X we can simply merge the ADG objects and halt further processing of X, since we know that Y cannot introduce any new paths that were not considered when processing X. On the other hand, if Y is not a subset, it is necessary to continue by examining the "union" of the substructure of objects in X with those in Y. We can reuse our incremental DataGuide maintenance algorithms (Figure 3.6) to minimize the amount of redundant work.

- How do we efficiently decide whether two sets are similar? Recent work has shown that we can efficiently determine whether two sets have a high percentage of elements in common [BGMZ97]. But the decision becomes more expensive as the threshold similarity percentage drops, since we cannot disqualify a potential match as quickly.

Similarity Threshold	Objects	Edges	False Paths
100% (Strong)	273	366	-
95%	241	308	0
90%	214	256	2
80%	200	241	6
70%	170	238	9
50%	117	140	9
30%	115	140	9
15%	110	138	9
1%	65	124	21

Table 3.3: Object-matching ADGs for the DBGroup database

3.7.2 Object Matching Experiments

For our experiments, we focused on the size and accuracy of the ADGs rather than absolute speed of construction, since for this work we used a simple, untuned B-tree-based data structure for computing set similarity.

We begin by testing the object matching approach over a larger version of the DBGroup database used in Section 3.3. This version contains about 3600 objects and 4200 edges. The database is highly cyclic, and while the overall structure is regular there are many " islands" of irregularity and incompleteness. The first row of Table 3.3 shows the size of the strong DataGuide. The remaining rows show the different ADG sizes for varying similarity threshold percentages. Quantifying the level of approximation is a challenge. As one simple metric, we counted how many false paths appear in the ADG but not in the strong DataGuide. Using depth-first search, once we determine that a path p is false, we do not continue to count paths for which p is a prefix, since of course they will be false as well.

Next, we attempted to analyze a 4MB subset of the Internet Movie database (www.imdb.com), a highly cyclic semistructured database with information about movies, actors, directors, producers, writers, etc. The database has about 60,000 objects and 95,000 edges. Unfortunately, our strong DataGuide algorithm did not terminate before exhausting resources: because of certain kinds of database cycles, the algorithm generated many very long paths (over 1000 labels) without finding a repeated target set. We were hopeful that the ADG would perform better, but unfortunately we hit the same problem. The algorithm generated too many small, nearly disjoint target sets that did not merge. Thus, while object-matching ADGs are efficient and effective when the number of target sets is manageable, the algorithm still is too expensive for certain larger, cyclic databases.

3.7.3 Role Matching

Rather than approximating DataGuides based on target sets, another approach is to merge DataGuide objects based on label paths (*roles*). More formally, we consider building ADGs based on Boolean *path merging* functions. If such a function $M(p_1, p_2)$ returns *True* for label paths p_1 and p_2, then paths p_1 and p_2 will point to the same ADG object. We discuss two possible merging functions.

Suffix Matching

In basic *suffix matching*, the merging function $M(p_1, p_2)$ returns *True* if and only if the last labels of p_1 and p_2 are the same. This approximation restricts the ADG to have one object per label. Figure 3.14(a) shows a sample database fragment, and its suffix-matching ADG is shown in Figure 3.14(b). Note that a suffix-matching ADG is essentially the same as the *1-Representative Object* described in [NUWC97], the original Stanford paper on which DataGuides were based.

The suffix-matching ADG is straightforward to characterize. While we can create it with a merging variant of the DataGuide construction algorithm, a simpler method is just to build a hash table: for each label l, we store information about all of the labels that directly follow l in the database. For the final step, we can construct the ADG by identifying the root label and walking what is essentially an adjacency-list graph representation inside the hash table. Construction time is at worst quadratic in the size of the database, since building the hash table requires examination of all paths of length 2.

The suffix-matching ADG is very effective when each label consistently identifies the "same type" of object. As one example where it could be problematic, consider our DBGroup database, where the Author label is used to identify both the authors of group publications and authors of members' favorite books. The suffix-matching ADG implies that Asimov, Salinger, and Kerouac may have Stanford email addresses! To help alleviate such problems, a natural extension to this approach is to match suffixes of length 2, 3, or k. As described in [NUWC97], we can generalize the hash-table approach to k-length suffixes, and that paper also proposes several algorithms for building more compact representations.

Database	Approximation	Objects	Edges	False Paths
DBGroup	Suffix	102	134	240
DBGroup	Path-cycle	240	317	3
Movies	Suffix	38	63	-
Movies	Path-cycle	76	96	-

Table 3.4: Role-matching ADGs

Path-cycle Matching

As an alternative to matching suffixes of a particular length, we consider a different path merging function that specifically addresses DataGuide performance problems caused by cyclic databases. Note that a strong DataGuide can have cycles itself when target sets are repeated along a path. But for larger databases, experience shows that paths grow to giant lengths before reaching an identical target set. As mentioned in Section 3.7.1, the problem persists even when we are willing to settle for similar target sets. Hence, we encode in a path merging function the following heuristic: if we see a specific label more than once along a path from the root, we assume that we have hit a "semantic" cycle and we merge the paths. For example, in our DBGroup database, suppose that at some point we create an ADG object for the path DBGroup.Paper. As we continue to explore this path, we create a new ADG object for DBGroup.Paper.Author, but when we encounter DBGroup.Paper.Author.Paper we assume that seeing Paper again indicates a schema cycle. Hence, we point back to the ADG object for DBGroup.Paper. This *path-cycle matching* ADG is shown in Figure 3.14(c); note that this approach avoids the suffix-merging problem of combining paper authors with group members' favorite authors.

Within our merging function framework, the path-cycle matching function $M(p_1, p_2)$ returns *True* if and only if p_1 is a prefix of p_2 (or p_2 is a prefix of p_1) and the last labels of p_1 and p_2 are the same.

3.7.4 Role Matching Experiments

For our experiments we modified the depth-first object-matching algorithm to merge ADG objects instead based on either suffixes or path-cycles. Table 3.4 shows experimental results for both types of approximations on the DBGroup and Movies databases introduced in Section 3.7.2. Again, the false paths column is in comparison to the strong DataGuide, which we were unable to generate for the Movies database.

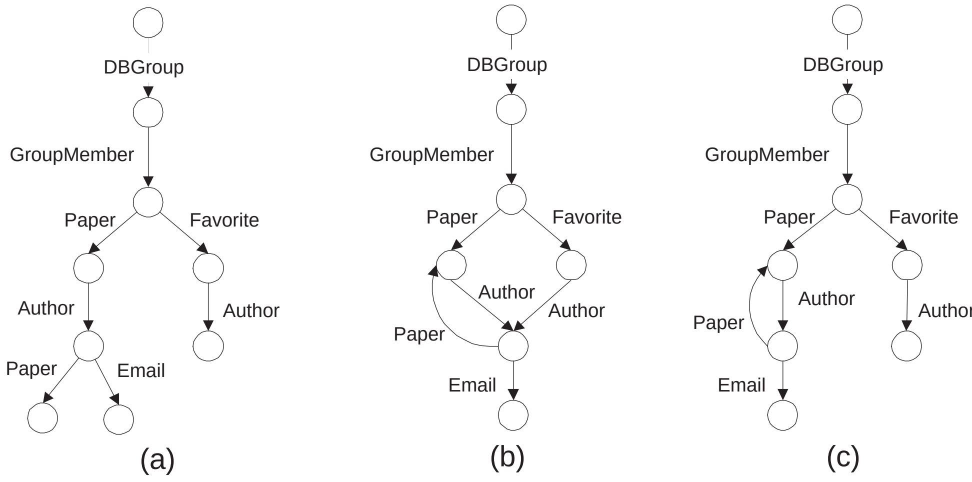


Figure 3.14: A sample OEM database, its suffix-matching ADG, and its path-cycle matching ADG

Note that the suffix approximation produced numerous false paths for the DBGroup database, many of which were due to the Author label problem described in Section 3.7.3. Using suffixes of length 2 would fix the problem for this database. Another interesting fact is that the smallest DBGroup object-matching approximation from Section 3.7.2 is actually smaller than the suffix-matching ADG, due to the fact that the database has many objects serving multiple roles (e.g., members as authors, project members, advisors, etc.). As could be expected, in both databases the path-cycle approximation is significantly larger than the suffix match. Perhaps the most striking results are the tiny sizes of the role matching approximations for the Movies database, given that we could not even build the strong DataGuide (or the object-matching approximation) for this database.

3.7.5 Summary

Since the space of possible semistructured databases is enormous and varied, it is difficult to choose the best approximation for every situation. Nonetheless, we can summarize the best and worst features of strong DataGuides and the Approximate DataGuides introduced in this section.

- *Strong DataGuides*: Strong DataGuides are always accurate and can be used as a path index, something for which we cannot use ADGs. For tree-structured, acyclic, or smaller cyclic databases, strong DataGuides usually perform well. For larger cyclic databases, it may be better to use an ADG. For path indexing, [MS99] proposes a graph-based path indexing structure that relaxes the DataGuide (and ADG) constraint of a database label path existing only once in the index. (It would be interesting to explore this approach in general, although for some DataGuide uses we do depend on a path existing only once, such as for the user interface.)

- *Object Matching*: This approach approximates a DataGuide based on objects having multiple incoming paths. Hence, it is an approximation for graph-structured databases only; for trees, a strong DataGuide is generated. An adjustable threshold lets the level of approximation be tuned. Unfortunately, however, the algorithm can still be prohibitively expensive for large, cyclic databases.
- *Suffix Matching*: The best feature of suffix matching is its predictable construction performance. The algorithm also is not biased to rooted paths— it provides information about path suffixes wherever they may appear in a database. Unfortunately, this approximation can yield skewed summaries and statistics if labels are used in different ways throughout a database. We can increase the suffix length that we match to increase accuracy, although doing so makes the algorithm more expensive.
- *Path-cycle Matching*: This approximation addresses problems caused by cyclic data without bias to paths of any specific length. Unfortunately, it is difficult to characterize just how computationally expensive this approach is.

Ultimately, the " best" ADG may depend on the database we are summarizing. Further, it may be possible to combine some of the above techniques, such as path-cycle and object matching.

3.8 Related Work

DataGuides were inspired by initial work at Stanford on *Representative Objects* [NUWC97]. This work proposed the importance of summarizing semistructured data via a NFA-to-DFA conversion. A representative object is defined as a function that can answer schema discovery questions about objects in a semistructured database. A DataGuide is an effective implementation of what is defined as a *Full Representative Object*. Work in [NUWC97] also focuses on enabling schema discovery when only considering paths of length k. Such functions are called *k-Representative Objects* (*k-ROs*). 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. Indeed, as mentioned in Section 3.7.3, a suffix-matching Approximate DataGuide is equivalent to the implementation of a *1-RO* suggested in [NUWC97].

Related work from [NAM98] gives algorithms for finding "approximate typings" of semistructured databases based on patterns of incoming and outgoing edges. In comparison, we are less concerned with extracting a set of object types; rather, our goal is to provide a structural summary that allows a semistructured database system (or a user of one) to quickly extract information about label paths in the database.

In other related work, [BDFS97] describes how we can define *graph schemas* for graph-structrured databases. However, their perspective is more traditional, since they assume that the database must still adhere to the schema, and they provide algorithms for testing whether a database adheres to a given schema. In contrast, DataGuides always conform to the database. However, choosing an effective Approximate DataGuide can be thought of as fitting a "good" graph schema to an existing database at a particular moment in time.

3.9 DataGuide Impact

Since their introduction in 1997, DataGuides have become recognized as a core concept within the field of semistructured data. For example, a recent book about semistructured data aimed at the mainstream technical community has numerous references to DataGuides [ABS99]. Further, DataGuides have served as a springboard for research outside Stanford, including the following projects.

- As mentioned in Section 3.7.5, Milo and Suciu have developed a new path indexing scheme that is related to our approach from Section 3.6, but takes advantage of a relaxed DataGuide definition to offer better performance [MS99].
- Liefke and Suciu leverage DataGuides to improve the performance of their XML compression tool, XMill [LS00].
- MiroWEB, a data integration project at the University of Versailles, cites DataGuides as the basis for their interactive query browser [BCSYDN+99].

Chapter 4

Interactive Query and Search of Semistructured Data

In this chapter we focus on the end-user's perspective of searching and querying a semistructured database. As more and more semistructured data is made available over the Web, it is important to enable casual Web users to interact effectively with the data. In Chapter 2 we introduced the Lorel query language for semistructured data. While Lorel is a powerful declarative language, like SQL it is too complicated for a casual end user to master. It is possible to handle certain queries by having users fill in hard-coded forms, but this approach by nature limits query flexibility. In Chapter 3 we showed how an interactive DataGuide simplifies the process of formulating a certain class of queries over a semistructured database. Still, even with a DataGuide, such an approach to querying semistructured data does not take into account two important characteristics of typical Web users:

- Users are comfortable initiating a search with simple keywords.
- Users find it natural to explore the results of an initial search, perhaps refining their search criteria iteratively until the desired information is found.

In this chapter we present a model for interactive query and search sessions over semistructured data that addresses these two points. First, we explain how we can implement searches based on a single keyword over a semistructured database. (To support searches based on multiple keywords, we rely on our *proximity search* techniques, described in detail in Chapter 5.) Second, to enable users to refine a search, we want to expose and summarize the structure of the database " surrounding" any query result. To create such a summary, we build dynamically and present to the user a

DataGuide that summarizes paths not from the database root (as in Chapter 3), but instead from the objects returned in the query result. A user can then repeat the process by submitting a query from this "focused" DataGuide or specifying additional keywords, ultimately locating the desired results.

In the context of the query functionality matrix from Chapter 1, the contributions of this chapter, along with Chapter 5 on proximity search, fill in Entry 4: enabling keyword-based search over semistructured databases (Section 1.1.5).

Our discussions in this chapter are again in the context of the *Lore* project (Chapter 2), involving OEM, Lorel, and DataGuides (Chapter 3). However, our results are applicable to other similar graph-based data models (e.g., [BDHS96]), as well as to XML [XML98].

In the rest of the chapter, Section 4.1 presents a simple motivating example to illustrate why new functionality is needed in a semistructured database system to support interactive query and search. Our session model is described in Section 4.2, followed by three sections covering the new required technology:

- Keyword search (Section 4.3): Efficient data structures and indexing techniques are needed for quickly finding objects in a semistructured database that match keyword search criteria. While we may borrow heavily from well-proven information retrieval (IR) technology, the new context of a graph database is sufficiently different from a simple set of documents to warrant new techniques.

- DataGuide enhancements (Section 4.4): Computing a DataGuide over each query result can be very expensive, so we have developed new algorithms for computing and presenting DataGuides piecewise, computing more of the DataGuide as needed.

- Inverse pointers (Section 4.5): To fully expose the structural context of a query result, it is crucial to exploit *inverse pointers* (pointers to an object's parents) when creating the DataGuide for the result, browsing the data, and submitting refining queries. While support for inverse pointers may seem straightforward, the major proposed models for semistructured data are based on directed graphs, and inverse pointers have not been considered in the proposed query languages [AQM$^+$97, BDHS96, FFLS97]. Similarly, proposed query languages for XML also do not support inverse pointers [DFF$^+$99a, RLS98].

Many of the contributions of this chapter were first published in [GW98].

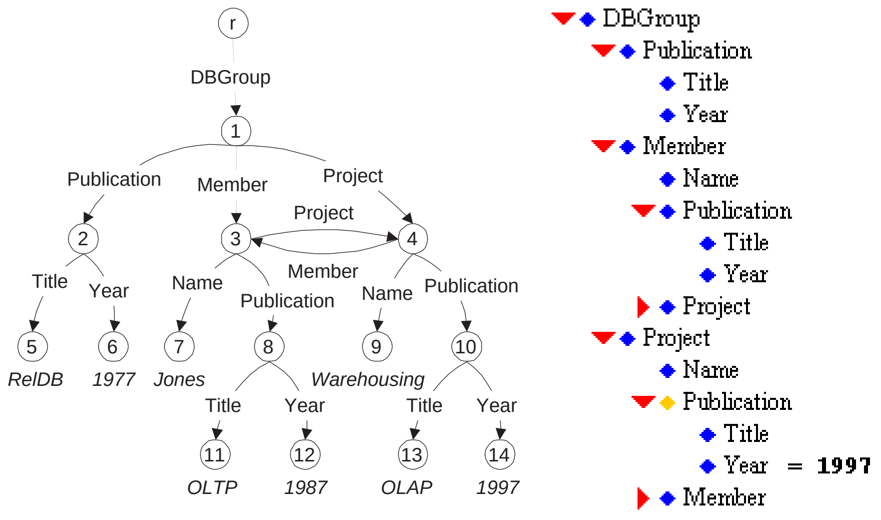


Figure 4.1: A sample OEM database and its DataGuide

4.1 Motivating Example

In Figure 4.1, consider a sample OEM database along with its DataGuide, as explained in Section 3.5. In this example, Publication objects appear in the database along several different paths: directly under DBGroup, under DBGroup.Member objects, and under DBGroup.Project objects. The DataGuide reflects a user-specifed query to select all *project* publications from 1997. The equivalent Lorel query is:

```
Select DBGroup.Project.Publication
Where DBGroup.Publication.Year = 1997
```

The result of this query is a singleton set containing object 10. More specifcally, when a query returns a result, a new object is created in the database with an incoming label Answer, and all objects in the query result are then made children of the Answer. [1] The new Answer edge is available as an entry point (as introduced in Section 2.2) into the database for successive queries, and the label for the children of Answer is deduced from the query— in this case, Publication.

Now suppose a user wishes to fnd all publications from 1997, a seemingly simple query. (Recall that our DataGuide query only found publications associated with projects.) It is possible to write a Lorel query with wildcards to fnd this result, but as discussed above, casual users will not want to

[1] Lorel queries may create more complicated object structures as query results, but for presentation purposes we do not consider such queries in this chapter; the approach and results in this chapter can easily be generalized.

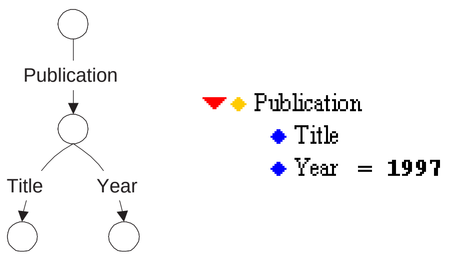


Figure 4.2: DataGuide constructed over result of finding all publications

enter a textual Lorel query. Suppose the user tries to use the DataGuide to locate the information. Even in this simple case, there are numerous paths to all of the publications; in a larger database the situation may be much worse. While the DataGuide does a good job of summarizing paths from the root, a user may be interested in certain data independent of the particular topology of a database.

In this situation, a typical Web user would be comfortable entering keywords: “Publication,” “1997,” or both. Suppose for now the user searches for “Publication” to get started. (We will address the case where the user searches for “1997” momentarily, and we discuss the issue of multiple keywords in Section 4.3.) If the system generates a collection of all Publication objects, the answer is objects $\{2, 8, 10\}$, again identified by the new edge Answer. While this initial result has helped focus our search, we really only wanted the Publication objects from 1997. One approach would be to browse all of the objects in the result, but in a realistic large database this may be difficult. Rather, we dynamically generate a DataGuide over the answer, as shown in Figure 4.2. Notice now that even though Title and Year objects were reachable along numerous paths in the original DataGuide, they are consolidated in Figure 4.2. As shown in the DataGuide, the user can mark Publication for selection and enter a filtering condition for Year to retrieve all 1997 publications. Getting the same result in the original DataGuide would have required three selection/filtering condition pairs, one for each possible path to a Publication.

The above scenario motivates the need for efficient keyword search, and for efficient (online) DataGuide creation over query results. Next, we show how these features essentially force a system to support inverse pointers as well. Suppose the user had typed “1997” rather than “Publication.” This time, the answer in our sample database is the singleton set $\{14\}$, and the DataGuide over the result is empty since the result is just an atomic object. This example illustrates that the user needs to see the area “surrounding” the result objects, not just their subobject structure as encapsulated by the DataGuide. Given a set of objects, we can use inverse pointers to present the “surrounding area” to the user; for example, we can give context to a specific Year object by showing that it is the child of a Publication object. By exploring both child and parent pointers of objects in a query result, we can create a more descriptive DataGuide.

4.2 Query and Search Session Model

Recall that our model is developed in the context of users interacting with Lore (Chapter 2). We define a Lore *session* over an initial database D_0, with root r and initial DataGuide $G_0(r)$, as a sequence of *queries* $q_1, q_2, \ldots, q_n$. A query can be a "by example" DataGuide query, a keyword search, or, for advanced users, an arbitrary Lorel query. The objects returned by each query q_i are accessible via a complex object a_i with entry point $Answer_i$. After each query, we generate and present a DataGuide $G_i(a_i)$ over the result, and users can also browse the objects in each query result. Perhaps counterintuitive to the notion of narrowing a search, we do not restrict the database after each query. In fact, the database D will grow monotonically after each query q_i. After q_i, $D_i = D_{i-1} \cup a_i$. Essentially, each DataGuide helps focus the user's next query without restricting the available data. In the following the three sections, we discuss three technologies that enable efficient realization of this model of interaction: keyword search, dynamic DataGuides, and inverse pointers.

4.3 Keyword Search

Defining and implementing keyword search over a semistructured database is a new problem. We begin by discussing how we can process a search based on a single keyword, and then we touch on the issue of how we handle ranked results. Chapter 5 focuses on new approaches for supporting searches over multiple keywords.

4.3.1 Single Keyword Search

In the IR arena, a search for a keyword typically returns a list of documents containing the specified keyword. In a semistructured database, pertinent information is found both in atomic values of type string (hereafter called "text objects") and in labels on edges. Thus, it makes sense to identify both text objects containing the specified keyword, and objects with incoming labels matching the keyword. For example, if a user enters "Publication," we would like to return all objects pointed to by a Publication edge, along with all text objects with the word "Publication" in their data. This approach is similar in spirit to the way keyword searches are handled by Yahoo! (yahoo.com). There, search results contain both the *category* and *site* matches for the specifed keywords.

Often, results to keyword searches are ranked according to some scoring function. For now, we assume that results of keyword searches are unranked; we will address the issue of ranking in Section 4.3.2.

While a keyword search over values and labels is expressible as a query in Lorel, we need to ensure speedy execution of keyword searches, so it is worthwhile to consider them as a special case. Since the number of unique labels in a database is typically small, we can use a naive search to find the labels that contain a given keyword; then, we can use a simple inverted-list index to identify all objects with a given incoming label. In contrast, locating atomic text objects that contain an arbitrary keyword expression is a larger challenge: we effectively need to build a full-text search engine that can match keyword expressions to database objects.

Rather than build our own specialized full-text search engine, we decided to leverage work in traditional text search engines, which match keywords to documents. In particular, we decided to integrate the Glimpse [MW93] search engine to provide full-text indexing support within Lore. Given a collection of documents, Glimpse builds indexes that enable fast regular expression searches over those documents, including simple keyword searches. The result of a Glimpse search is a set of <document identifer, offset> pairs, identifying the positions in documents that match the search expression. Our task was to exploit this interface to provide the somewhat different keyword-search functionality we needed in Lore.

Given the interface to Glimpse, we want to map a Lore keyword search into a Glimpse search, and translate Glimpse's results into a collection of Lore OIDs that represent the matching database objects. At one extreme, we could map each Lore object into a separate fle, but this approach could easily overwhelm the fle system. At the other extreme, we could map the entire Lore database into a single fle, along with additional information associating database objects with their positions within the fle. Of course, another option is to partition the database objects across any number of fles. Some initial experiments indicated that Glimpse was just as effective processing one large fle as it was with more, smaller fles. Hence, we developed our prototype by dumping each Lore database into a single fle for Glimpse to index.

In more detail, the following steps explain how we create the Glimpse-based text index for a Lore database.

- We create one text fle GlimpseData that will contain a sequential dump of all text objects in a Lore database.

- We create one text file GlimpseMap to serve as a map that allows us to translate Glimpse search results back into Lore OIDs.

- We traverse the entire Lore database. For each atomic text object o, we first output to GlimpseMap a record containing the current length of GlimpseData (representing the *offset* of the data for o) and the OID of o. Next, we output to GlimpseData the entire text of o.

- We build an ISAM [Wag73] index GlimpseIndex on top of GlimpseMap that takes as input an offset *inputOffset*, and returns the OID associated with the largest offset in GlimpseMap less than or equal to *inputOffset*.

- We delete GlimpseData; it is not needed since Glimpse builds its own indexes and any offsets returned as search results can be mapped to OIDs via our GlimpseIndex over GlimpseMap. (In our implementation of GlimpseIndex we must keep GlimpseMap.)

Glimpse takes as input a search expression. While Glimpse supports expressions containing more than one keyword, we have developed our own technology for effectively handling searches of multiple keywords across a database, as we will discuss in Chapter 5. Thus, we restrict our use of Glimpse to only single keywords.

After Glimpse processes its input *keyword*, we take the following steps to translate Glimpse's results into a set of matching Lore OIDs.

- First, we pass *keyword* to Glimpse.

- If Glimpse returns no "hits," then of course the result is empty. Otherwise, Glimpse returns a set of file offsets representing matches. (All matches are from the single GlimpseData file, so document identifers can be ignored.)

- For each returned offset, we use GlimpseIndex to translate the offset into a matching OID. We keep track of the OIDs of all objects that match *keyword*. If one object matches *keyword* more than once, we maintain that information as well.

- By default, Lore and OEM are set-based, so the final step of generating a query result is to produce a set of matching objects. While Glimpse results are not ranked, we discuss ranking briefly in Section 4.3.2.

With this approach, we leverage both the functionality and performance of an existing specialized search engine. We do have the overhead at index creation time of dumping the data into a text file, and at runtime we have some overhead of a system call to Glimpse and additional index lookups. Still, this approach allows Lore to be loosely coupled with Glimpse and incorporate future improvements or bug fixes to Glimpse. Further, it would require minimal effort to port our techniques to any other search engine that returns the file offsets of search expression matches within a document. If a search engine only returns matching document IDs, then we would need to map each database object to its own file. Alternatively, we could map multiple objects to a file, using search engine results as just the first phase of identifying potentially relevant objects.

4.3.2 Ranking Results in Lore

In environments that support keyword-based or other types of "fuzzy" search, ranking is an often critical component of the results. Through some kind of scoring function, system-generated rankings help a user sift through many potential matches and focus on the most important data. Typically, however, database management systems are based on sets or bags— data is either "in" or "out" of a query result, and there is no built-in notion of rank. Lorel queries are no exception, since the result is simply a collection of OEM objects. Smooth integration of rankings into such a model is a challenging problem, one we defer to future work (Chapter 8).

We do support rankings in our keyword searches (and in proximity search, described in Chapter 5), for which we developed a simple standalone *keyword search interface* for Lore. Through this interface, we can return ranked results instead of just a flat collection of objects. However, because this interface is separate from Lorel, these ranked results are currently unavailable for follow-on queries.

Glimpse itself does not rank its results, so we assign a score to each matching object based on the amount of text in the object, the size of the text that matches the search, and/or the number of matches within the object. Currently, we use a simple score based on dividing the total amount of matched text in the object by the total size of the object.

4.4 DataGuide Enhancements

As described in the motivating example of Section 4.1, we wish to build DataGuides over query results, quickly enough to support interactive sessions. For this section, let us temporarily ignore the issue of inverse pointers. As shown in Section 3.3, computing a DataGuide can be expensive:

the worst case running time is exponential in the size of the database, and for a large database even linear running time would be too slow for an interactive session. We thus introduce two techniques to improve the running time of interactive DataGuide creation.

First, we can exploit the auxiliary data structures that are built to provide incremental DataGuide maintenance: targetHash and objectHash from Section 3.4. These structures guarantee that, when constructing the DataGuide for a query result, we never need to recompute a "sub-DataGuide" that has previously been constructed. In Figure 4.1, suppose a user searches for all "Projects," a query that would return the singleton set $\{4\}$. In this case, the DataGuide over $\{4\}$ is the same as the sub-DataGuide reachable along DBGroup.Project in the original DataGuide. We can dynamically determine this fact with a single lookup of $\{4\}$ in targetHash (see Section 3.4), and no additional computation is needed.

Second, we observe that an interactive user will rarely need to explore the entire DataGuide. Our experience shows that even in the initial DataGuide, users rarely explore more than a few levels. Most likely, after a reasonable "focusing" query, users will want to browse the structure of objects near the objects in the query result. Hence, for interactive sessions we modified the original depth-first DataGuide construction algorithm (Figure 3.4) to instead work breadth-first, and we changed the algorithm to build the DataGuide "lazily," i.e., a piece at a time. From the user's perspective, the difference is transparent except with respect to speed. When a user clicks on an arrow for a region that hasn't yet been computed, behind the scenes we send a request to Lore to generate and return more of the DataGuide. Our maintenance structures make it easy to interrupt DataGuide computation and continue later with no redundant work.

4.5 Inverse Pointers

Directed graphs are a popular choice for modeling semistructured data and XML, and the proposed query languages [DFF+99b, AQM+97, BDHS96] are closely tied to this model. Powerful regular expressions in the languages traverse forward pointers (children), but essentially no language support has been given to traversing inverse (parent) pointers. As our motivating example in Section 4.1 demonstrates, a parent may be just as important as a child for locating relevant data during an interactive session.

In this section we describe a simple modification to the Lore data model that makes access to inverse pointers as seamless as possible: for an object O with an incoming label "X" from another object P, we conceptually also make P a child of O via an edge with the new special label "XOf."

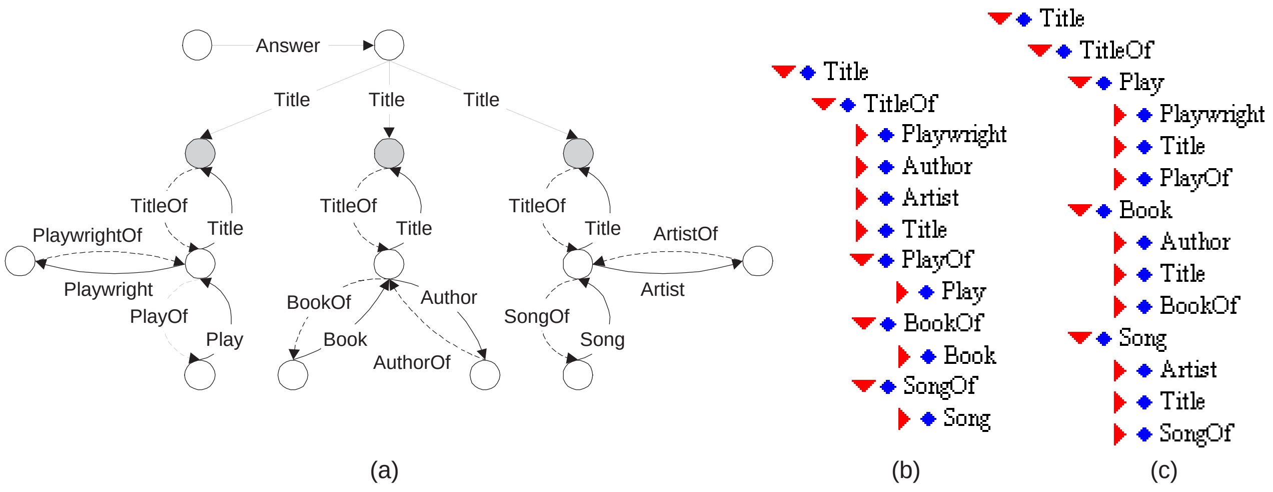


Figure 4.3: An OEM query result and two potential DataGuides

(Physical implementation of inverse pointers depends on the underlying data store; in Lore, for example, an auxiliary hash-based index keeps track of the parents of every object.) With this approach of using special Of labels, inverse pointers can be treated for the most part as additional forward pointers. For example, the Lorel language can support references to these special Of labels without any modification. However, in the context of DataGuides, the presence of inverse pointers may lead to some counterintuitive results— a topic we now address.

As motivated in Section 4.1, we wish to extend DataGuides to summarize a database in all directions, rather than only by following forward pointers. If the "Of" edges described above are simply added to the database graph, then we need not even modify our DataGuide algorithms. Unfortunately, this approach can yield some strange results. In OEM and most graph-based database models, objects are identified by their incoming labels. A "Publication," for example, is an object with an incoming Publication edge. This basic assumption is used by the DataGuide, which summarizes a database by grouping together objects with identical incoming labels. An "Of" edge, however, does a poor job of identifying an object. For example, given an object O with an incoming TitleOf edge, we have no way of knowing whether O is a publication, book, play, or song. Therefore, a DataGuide may group unrelated objects together.

As a more detailed example, suppose a user's initial search over a library database finds some Title objects. Figure 4.3(a) shows three atomic objects in the result (shaded in the figure), with dashed "Of" edges to show their surrounding structure. Figure 4.3(b) shows the standard DataGuide over this Answer. The problems with 4.3(b) should be clear: the labels shown under TitleOf are confusing, since the algorithm has grouped unrelated objects together. Further, the labels directly under TitleOf do not clearly indicate that our result includes titles of books, plays, and songs.

To address the problem, we have modified the DataGuide algorithm slightly to decompose further all objects reachable along an "Of" edge based on the non-"Of" edges to those objects. In particular, we temporarily transform the source graph such that an Of edge that originally points from object X to object Y is modified to point to a new node XY; further, for each incoming edge to Y with a non-Of label L, we add a new edge with label L from XY to Y. Figure 4.3(c) shows the DataGuide built over this transformed graph— a result we call a *Panoramic DataGuide*. In this DataGuide, we can see that TitleOf leads to a new intermediate object (whose target set is all of the new nodes in the transformed graph created for the TitleOf edges). Under TitleOf in the DataGuide, we can see that the Play, Book, and Song subobjects are now separated, yielding a more intuitive DataGuide for browsing. Of course, since OEM databases can have arbitrary labels and topologies, we have no guarantees that a Panoramic DataGuide will be the ideal summary; still, in practice it seems appropriate for many OEM databases. Note that adding inverse pointers to DataGuide creation adds many more edges and objects than in the original DataGuide, making our support for "lazy" DataGuides (Section 4.4) even more important.

As an alternative to using inverse pointers, a semistructured database system could "remember" the (forward) path traversed to evaluate a query. For example, consider the simple query from Section 4.1 to find all project publications from 1997. During query execution, the system could remember the path from the root of the database used to reach the results objects— in this case, the single DBGroup.Project.Publication path passing through objects 1, 4, and 10. The user could then explore this path to see some of the result's context. Lore can in fact provide such a *matched path* for each query result. However, when an execution strategy does not involve navigating paths from the root, generating a matched path from the root would drastically increase query execution time. Further, a matched path still does not allow a user to arbitrarily explore the database after a query result.

4.6 Related Work

For several of the topics covered in this chapter, such as modelling interactive query and search sessions and exploiting inverse pointers, we know of no previous related work. The problem of supporting keyword-based search in a database system has been addressed primarily at the level of integrating document collections with relational database systems, such as supporting keyword-search over specifc table felds known to contain documents [Ora99] or using virtual tables to model a search engine within the context of a SQL query [DM97, CDY95, GW00].

Chapter 5

Proximity Search in Databases

In Chapter 4 we discussed the semantics and implementation of keyword-based searches in Lore when the search term is a single keyword. Performing a keyword-based search over a semistructured database becomes more interesting and challenging when searching for multiple keywords. In a typical information retrieval (IR) setting, a search for two or more keywords identifies documents containing all keywords "close" together— at the least, both keywords must be in the same document, and often, a document is considered a "better" match if the keywords are near each other in the document text. The *near* operator is used in IR systems to perform explicitly such a *proximity search*: searching for keywords close to each other within a document [Sal89].

In a graph-structured database, textual distance can be a poor measure of the relationship between keywords. In XML, for example, nesting structure can be far more important in determining the "nearness" of document elements than the textual distance between them. As we saw in Chapter 1, in an XML representation of a movie database, two actors in the same movie will both be subelements of a specific movie element. However, in a textual representation of the database, the last actor of one movie may actually be listed closer to a different movie element he has no relation to. In a graph-structured database, measuring distance between objects or elements based on their relationship within the database structure is more meaningful than measuring text-based distance in a serialization of the database.

In this chapter we apply the general notion of proximity search to search across an entire database for objects that are "near" other relevant objects. We consider a graph-based data model such as OEM (Chapter 2), and proximity is defined based on shortest paths between objects. We demonstrate how proximity search enables interesting multiple-keyword searches over a semistructured database with intuitive results. Our approach to proximity search requires only that data can be viewed as an interconnected graph— it need not be "semistructured," nor need it be stored in a database for semistructured data. As we will show in Section 5.2, it is quite straightforward to view relational or object-oriented data as a graph. Thus, even though the implementation of our proximity search engine works within the Lore system, the techniques apply to traditional structured data as well as they do to "true" semistructured data. Referring back to our query functionality matrix (Section 1.1.5), our proximity search engine enables keyword-based search over both semistructured databases (Entry 4) and traditional databases (Entry 2).

Implementing proximity search in a graph-based database is significantly different from the traditional IR approach. Traditionally, keyword proximity is measured along a single dimension (text), and search is performed inside each text object. It is easy to compute the distances between words if we simply record the position of each word along this one dimension. In a graph-based database, we measure distance as the length of the shortest path between data objects. For efficient inter-object proximity search, we need to build an index that gives us the distance between *any* pair of database objects. Since there can be a huge number of objects, computing this index can be very time consuming. For traditional proximity search, on the other hand, we only need to know the distance between words within a single object, a much smaller problem. In this chapter we describe optimizations and compression schemes that allow us to build indexes that can efficiently report distances between any pair of objects. Experiments show that our algorithms have modest time and space requirements and scale well.

In Section 5.1, we provide a concrete example to further motivate our notion of proximity search in graph-based databases. Section 5.2 then defines our problem and framework in more detail. In Section 5.3, we describe how we have built proximity search into Lore to support multiple-keyword search. Section 5.4 details our algorithms for efficient computation of distances between objects, and experimental results are given in Section 5.5. We discuss related work in Section 5.6.

The basis of this chapter originally appear in [GSVGM98].

5.1 Motivating Example

The Internet Movie Database (www.imdb.com) is a popular Web site with information about more than 140,000 movies and 500,000 film industry workers. Regardless of whether the data is stored natively as a semistructured XML or OEM database, in a relational database, or in an object-oriented database, we can view the database as a set of linked objects, where the objects represent movies, actors, directors, and so on. In this application it is very natural to define a distance function based on the links separating objects. For example, since John Travolta stars in the movie "Primary Colors," there is a close relationship between the actor and the movie; if he had directed the movie, the bond might be tighter.

Within our framework, proximity searches are specifed by a pair of queries, each of which can be any type of query that returns a set of objects:

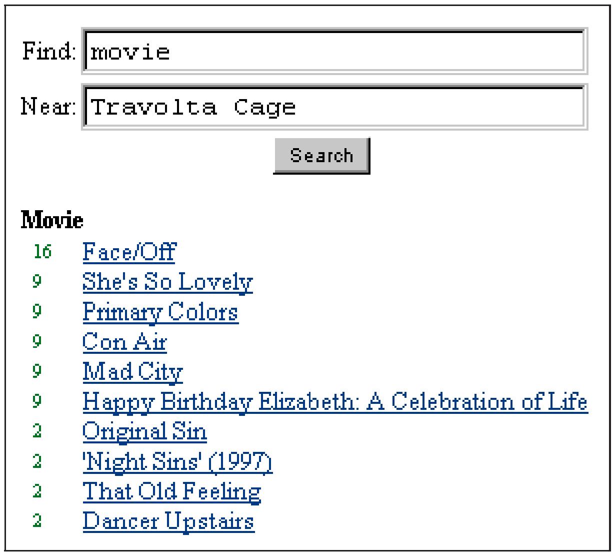
- A *Find query* specifes a *Find set* of objects that are potentially of interest. For our example, let us say that the fnd query is keyword-based. For instance, "Find movie" locates all objects of type "movie" or objects with the word "movie" in their body. In the Lore system, such a keyword search would use the techniques described in Section 4.3.1.

- Similarly, a *Near query* specifes a *Near set*. The objective is to rank objects in the *Find* set according to their distance to the *Near* objects. For our examples we assume the near query is also keyword-based.

For example, suppose a user is interested in all movies involving both John Travolta and Nicolas Cage. This query could be expressed as "Find movie Near Travolta Cage." Notice that this query does not search for a single "movie" object containing the "Travolta" and "Cage" strings. In this database, the person named "Travolta" is represented by a separate object, and similarly for "Cage." Movie objects simply contain links to other objects that defne the title, actors, date, etc. Thus, the proximity search looks for "movie" objects that are somehow closely connected to "Travolta" and/or "Cage" objects.

To illustrate the effect of this query, we show results of issuing the query over a version of the Internet Movie Database (IMDB) containing information about all 1997 flms, stored in OEM format within Lore (Chapter 2). Our Lore implementation of proximity search is described in more detail in Section 5.3. Figure 5.1 shows the query "Find movie Near Travolta Cage" along with the top 10 results. As we might expect, "Face/Off" scored highest since it stars both actors. That is, both actor objects are a short distance away from the "Face/Off" movie object. The next fve movies all received the same second-place score, since each flm stars only one of the actors. (See Section 5.2 for a detailed explanation of how ranking works.) The remaining movies refect indirect affliations— that is, larger distances. "Original Sin," for example, stars Gina Gershon, who also played a part in "Face/Off."

To illustrate other queries, a user could issue "Find movie Near Colorado" to locate all movies flmed in Colorado (or with the word "Colorado" in their titles). A user might issue "Find love Near comedy" to fnd all references to "love" in a comedy— movie titles, actor names, trivia, etc. As a


Find: movie

Near: Travolta Cage

Movie

16	Face/Off
9	She's So Lovely
9	Primary Colors
9	Con Air
9	Mad City
9	Happy Birthday Elizabeth: A Celebration of Life
2	Original Sin
2	'Night Sins' (1997)
2	That Old Feeling
2	Dancer Upstairs

Figure 5.1: Results of proximity search over the Internet Movie Database

final example, we might wish to rank movies by the number of different locations where they were filmed at by issuing " Find movie Near location." Our prototype is available to the public on the Web, as described in Section 5.3.

Proximity searches are inherently fuzzy. If one can precisely describe the desired information (e.g., what relation it occurs in, the exact path to it, the precise contents of fields) then traditional database queries will usually be best. Still, proximity search is very useful when it is impractical to generate a specific query, or when a user simply wants to search based on the general relevance of different data objects and then focus in on relevant data, as with the interactive query and search model discussed in Chapter 4.

No current database or IR systems provide general proximity search across interrelated objects. Often, applications implement particular versions of proximity search. For example, the IMDB Web site does offer a form for searching for movies with multiple specified actors. Our goal is to provide a general-purpose proximity service that can be implemented on top of any type of database system, semistructured or otherwise. Further, we demonstrate that our techniques are practical by implementing a proximity search engine within Lore.

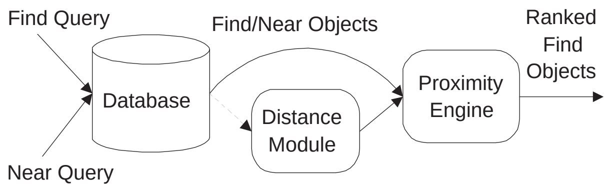


Figure 5.2: Proximity search architecture

5.2 The Problem

The problem, expressed in its most general terms, is to rank the objects in one given set (the *Find* set) based on their proximity to objects in another given set (the *Near* set), assuming objects are connected by given numerical " distances." We first discuss our conceptual model in detail, and then we formalize our notion of proximity.

5.2.1 Conceptual Model

Figure 5.2 shows the components of our model. An existing database system— whether it be semistructured, relational, or object-oriented— abstractly stores a set of data objects. Applications generate *Find* and *Near* queries at the underlying database. Note that while our motivating example used keyword-based search to identify the *Near* and *Find* sets, our framework is very general and is open to other types of queries: it is designed simply to relate two sets of objects based on proximity in a graph.

The database evaluates the queries and passes *Find* and *Near* object result sets, which may themselves be ranked, to the *Proximity Engine*. (For example, keyword search in Lore returns a ranked list of objects as a result, as described in Section 4.3.) Database objects are opaquely exported to the Proximity Engine, which only deals with object identifers (OIDs).[1] The Proximity Engine then re-ranks the *Find* set, using distance information, and possibly taking into account the initial ranks of the *Find* and *Near* objects. The distance information is provided by a *Distance Module*. Conceptually, the *Distance Module* is a black box that provides to the Proximity Engine a set of triplets (X, Y, d), where d is the distance between " adjacent" database objects with identifers X and Y. (Note that the distance module uses the same identifers as the database system.) In Lore,

[1] Most relational systems do not expose explicit row identifers; we can use primary key values or " signatures," e.g., checksums computed over all tuple feld values.

the Distance Module is simply the weighted edges that connect objects in the original database, as we will discuss further in Section 5.3. The Proximity Engine then uses these base distances to compute the lengths of shortest paths between all objects. Because we are concerned with " close" objects, we will compute the distance between any two objects exactly only up to some constant K, returning ∞ for all distances greater than K. This assumption enables improved algorithms, as described in Section 5.4.

To the Proximity Engine, the database is simply an undirected graph with weighted edges. In our motivating example, the underlying database is indeed a Lore database. Still, our proximity calculations work over traditional database systems as well, as long as the data is exported as a graph-structured view. For example, the database system may be relational, as illustrated by the left side of Figure 5.3, which shows a small fragment of a normalized relational schema for the Internet Movie Database. The right side of the figure shows how that relational data might be interpreted as a graph by the Proximity Engine. Each Movie and Actor tuple is broken into multiple objects: one object for the tuple and additional objects for each attribute value. Distances between objects are assigned to reflect their semantic closeness. For instance, in Figure 5.3 we assign small weights (indicating a close relationship) to edges between a tuple and its attributes, larger weights to edges linking tuples related through primary and foreign keys, and the largest weights to edges linking tuples in the same relation. (For clarity, the graph shows directed, labeled edges; our algorithms ignore the labels and edge directions.) Of course, the distance assignments must be made with a good understanding of the database semantics and the intended types of queries. It is simple to model object-oriented, network, or hierarchical data in a similar manner.

Currently, our prototype implementation of the proximity engine is implemented as an indexing module within the Lore system, though it could be implemented as a separate component that works over OIDs exported from any database system.

5.2.2 Proximity and Scoring Functions

Recall that our goal is to rank each object f in a *Find* set F based on its proximity to objects in a *Near* set N. Each of these sets may themselves be ranked by the underlying database system. We use functions r_F and r_N to represent the ranking in each respective set. We assume these functions return values in the range $[0, 1]$, with 1 representing the highest possible rank. We define the distance between any two objects $f \in F$ and $n \in N$ as the weight of the shortest path between them in the underlying database graph, referred to as $d(f, n)$. To incorporate the initial rankings as well, we define the *bond* between f and n ($f \neq n$):

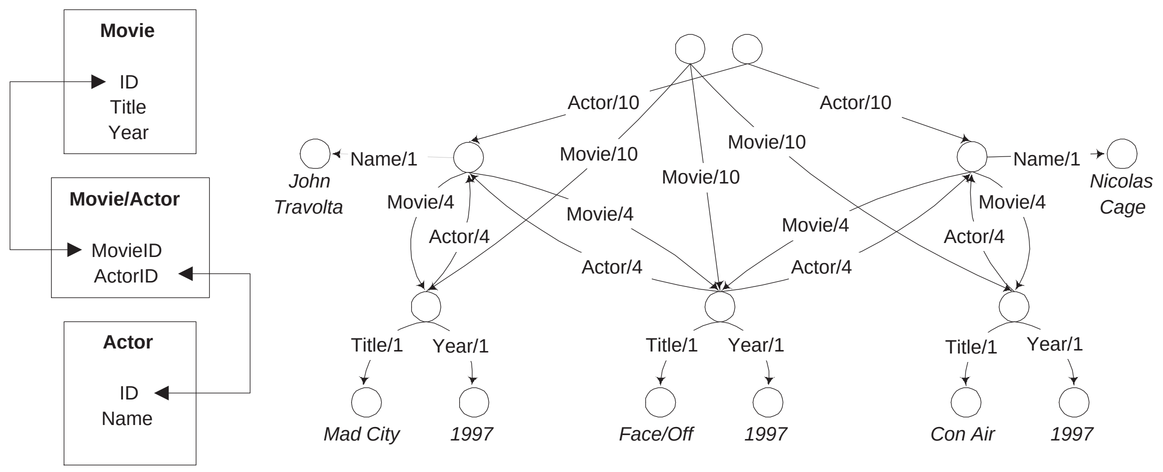


Figure 5.3: A fragment of the movie database relational schema and a database instance as a graph

$$b(f,n) = \frac{r_F(f)r_N(n)}{d(f,n)^t} \tag{5.1}$$

(We set $b(f,n) = r_F(f)r_N(n)$ when $f = n$.) A bond ranges from $[0,1]$, where a higher number indicates a stronger bond. The tuning exponent t is a non-negative real that controls the impact of distance on the bond.

While a bond reflects the relationship between two objects, in general we wish to measure proximity by scoring each *Find* object based on all objects in the *Near* set. Depending on the application, we may wish to take different approaches for interpreting bonds to the *Near* objects. We discuss three possible scoring functions:

- *Additive*: In the query from our motivating example to "Find movie Near Travolta Cage," (Section 5.1), our intuition leads us to expect that a film closely related to both actors should score higher than a film closely related to only one. To capture this intuition, we score each object f based on the sum of its bonds with *Near* objects:

$$score(f) = \sum_{n \in N} b(f,n) \tag{5.2}$$

Here the score can be greater than 1.

- *Maximum*: In some settings, the maximum bond may be more important than the total number. Thus, we may define
$$score(f) = \max_{n \in N} b(f, n) \tag{5.3}$$
In this case, scores are always between 0 and 1.

- *Beliefs*: We can treat bonds as beliefs [Goo61] that objects are related. For example, suppose that our graph represents the physical connections between electronic devices, such that two objects close together in the graph are close together physically as well. Assume further that r_N gives our belief that a *Near* device is faulty (1 means we are sure it is faulty). Similarly, r_F can indicate the known status of the *Find* devices. Then, for a device $f \in F$ and a device $n \in N$, $b(f, n)$ may give us the belief that f is faulty due to n, since the closer f is to a faulty device, the more likely it is to be faulty. Our belief that f is faulty (between 0 and 1), given the evidence of all the *Near* objects, is:
$$score(f) = 1 - \prod_{n \in N} (1 - b(f, n)) \tag{5.4}$$

Of course other scoring functions may also be useful, depending on the application. We expect that proximity search engines can provide several "standard" scoring functions, and that users submitting queries will specify their intended scoring semantics. This approach is analogous to how users specify what standard function (e.g., COUNT, MAX, AVG) to use in a statistical query. In the Lore implementation, we currently only use the additive scoring function.

5.3 Lore Implementation

We implemented our proximity architecture and algorithms within the Lore database management system (Chapter 2). Proximity search extends the keyword searching facility described in Section 4.3 to provide effective search using multiple keywords, across an entire database. Any OEM database can serve as input to our proximity engine, with one slight modification: we enable Lore to store weights on edges. Without this feature, experiments showed that the "diameter" of a typical database was just too small; in other words, there was little variation in distances and too many unrelated objects ended up "tying" in proximity measurements. By adding weights, we can emphasize the intended strength of relationships indicated by edges in the database. Each edge in the database may be assigned a specific weight; alternatively, weights may be specified according to incoming labels or label paths. As a simple example, we could specify that all edges labeled Actor should be assigned a certain weight. Note also that our proximity search engine ignores the directionality of an OEM graph: the distance from a parent P to its child C always is the same as the distance from C to P.

In our Lore implementation, we generate the *Find* and *Near* sets using the *keyword search interface* described in Section 4.3. Recall that in an OEM database, a keyword can identify an object with a specific incoming edge label, an atomic text object whose data contains the keyword, or both. The two " Category" drop-down menus in Figure 5.1 provide an alphabetical list of unique labels in the database; the number of unique labels is generally small, and the list can be very helpful for specifying meaningful searches. Choosing a label from either menu adds that label as a keyword in the corresponding field. For each keyword, we execute a ranked, single-keyword search as described in Section 4.3; we add all matching objects, with their rankings, to the *Find* or *Near* set as appropriate. Note that if multiple keywords are used to generate a *Find* set (or *Near* set), then we generate the appropriate set by performing single-keyword search on each keyword and computing the union of all matching objects; currently this case is not an interesting application of using multiple keywords since the relationship is expected to exist across the two *Find* and *Near* sets, not within them.

Based on informal usability tests, we chose to set tuning parameter t to 2 in our bond definition (Equation 5.1), to weight nearby objects more heavily; this setting causes a bond to drop quadratically as distance increases. We use the additive scoring function (Equation 5.2) to score each *Find* object. Together, our choice of tuning parameter and scoring function will give a *Find* object f_1 that is 1 unit away from a *Near* object n_1 twice the score of a *Find* object f_2 that is 2 units away from two *Near* objects n_2 and n_3. In the user interface, we linearly scale and round all scores to be integers.

Figure 5.4 summarizes the results of several proximity search queries over our DBGroup database, describing the members, projects, and publications of the Stanford Database Group, as used throughout this thesis. The database has been built from scratch in OEM, containing about 4200 objects and 3600 edges. Initial supplied distances are similar to those shown in Figure 5.3; for example, a root object is connected to all publications, projects, and group members via edges of weight 10, publications are connected to their titles via edges of weight 1, and group members are connected to their publications via edges of weight 4. Examples show that proximity search is a useful complement to traditional database queries, allowing users to narrow in on relevant data without having to understand the nature of all database relationships, and without fully specifying structural queries.

Find picture Near China	Photos of 6 Chinese students, followed by Prof. Widom, who advises 3 of them, and Prof. Ullman, who advises 2
Find publication Near Garcia	All of Prof. Garcia-Molina's publications, followed by publications of his students
Find publication Near Garcia Widom	The top publications are co-authored by Profs. Garcia-Molina and Widom, followed by their individual papers
Find group_member Near September	The top results are members born in September
Find publication Near OEM	The top pub. has "OEM" in its title, followed by a pub. stored in "oem.ps," followed by one with keyword "oem"

Figure 5.4: Summary of Stanford Database Group keyword searches

At the same time, proximity search provides more expressive power and useful results than the simple single-keyword search introduced in Chapter 4. Note that even in the Lore context this implementation reflects just one particular set of choices for instantiating our proximity model— how we generate the *FindNear* sets, our initial ranking functions r_F and r_N, our tuning exponent t in the bond definition, and our choice of scoring function.

5.4 Computing Object Distances

For our proximity computations to be practical, we need to find the distances between pairs of objects efficiently. In this section we discuss the limitations of naive strategies and then focus on our techniques for generating indexes that provide fast access at search time.

First, we discuss the framework for our distance computations. As described in Section 5.2.1, the proximity engine takes as input *Find* and *Near* sets of OIDs, and a set of base distances between adjacent objects. Let V be the set of objects. We assume the distances are provided by the Distance Module of Figure 5.2 as an *edge-list* relation E_1, with tuples of the form $\langle u, v, w\rangle$, if vertices $u, v \in V$ share an edge of weight w. For convenience, we assume that E_1 contains $\langle u, v, w\rangle$, if $\langle v, u, w\rangle$ is in E_1. Let G refer to the graph represented by E_1.

In graph G, we define $d_G(u, v)$ to be the shortest weighted distance between u and v. (We will drop the subscript G if it is clear which graph we are referring to.) As mentioned in Section 5.2.1, our proximity search focuses on objects that are "close" to each other. Hence, we assume all distances larger than some K are treated as ∞. In our prototype, setting $K = 12$ for the IMDB and DBGroup databases yields reasonable results, given the initial supplied distances.

5.4.1 Naive Approaches

At one extreme, we could answer a distance query by performing all required computation at search time. A classical algorithm to compute the shortest distance between two vertices is Dijkstra's single-source shortest path algorithm [Dij59]. The algorithm produces the shortest distance using a "best-first" search to compute shortest paths. At each iteration, we explore $N(v)$, the vertices adjacent to some vertex v. While the algorithm is efficient for graphs in main memory, exploring $N(v)$ may require $|N(v)|$ random seeks for an arbitrary disk-based graph, and computing the shortest distance could take as many as $|E_1|$ random seeks. Note that this behavior persists even when we are only interested in distances no larger than K. Further, since a general *FindNear* query requires multiple distance computations, we would have to call the algorithm $\min(|Find|, |Near|)$ times. (Each call to the algorithm finds the shortest path from a single given vertex to all others; we must therefore run the algorithm over each vertex in the *Find* set or each vertex in the *Near* set.)

At the other extreme, we could precompute shortest distances between all pairs of vertices and store them in a lookup table for fast access. The classical algorithm to compute all-pairs shortest distances is Floyd-Warshall's dynamic programming based algorithm [Flo62]. An obvious disk-based extension of the algorithm requires $|V|$ scans of G. The algorithm could be redesigned to be more efficient on disk, and this approach is similar to the algorithm we introduce in the next section. There has been much work on the related problem of computing the transitive closure of a graph. In Section 5.6 we discuss these approaches and why they are not suitable for our problem.

In the next section, we propose an approach for precomputing all-pairs distances of at most K that is efficient for disk-based graphs, using well-known techniques for processing "self-joins" in relational databases. Section 5.4.3 shows how we can exploit available main memory to further improve both the space and time requirements of index construction.

5.4.2 Precomputing Distances Using "Self-Joins"

We compute shortest paths between nodes up to some maximum distance by joining an input edge-list relation with itself, joining that result with itself, and so on. We begin with a high-level, intuitive explanation of our approach and then provide the actual algorithm. Given an input edge-list E_1 as introduced in Section 5.4, we join E_1 with itself by finding tuples that share exactly one vertex in common: if we find $\langle v_i, v_j, w_k\rangle$ and $\langle v_i, v'_j, w'_k\rangle$ in E_1, where $v_j \neq v'_j$, then we produce an output tuple of the form $\langle v_j, v'_j, w_k + w'_k\rangle$. Intuitively, by performing this step we are asserting that the distance from v_j to v'_j is no greater than $w_k + w'_k$.

Let us create a temporary relation E'_2 that contains all tuples in E_1 in addition to any joined tuples as above (i.e., tuples of the form $\langle v_j, v'_j, w_k + w'_k\rangle$). Note that for vertices v_j and v'_j, E'_2 contains tuples representing the distances of all possible paths between those two vertices through at most one other vertex. However, since our goal is to compute minimum distances, we ultimately want only the shortest distance between v_j and v'_j. We call E'_2 an *nonreduced* edge-list, since it may contain many tuples representing different distances between vertices. We *reduce* E'_2 by extracting only the tuple representing the minimum distance for any given vertex pair, and inserting this tuple into E_2. We call E_2 a *reduced* edge-list. For any two vertices v_i and v_j, E_2 has at most one tuple representing the shortest path between them through at most one other vertex.

We could perform a self-join on E_2 and continue repeating the process to identify eventually the shortest paths between all pairs of vertices. However, remember that we are only interested in shortest paths up to some K. Thus, during the self-join of E_1, we will add an extra condition that the sum of the weights of both edges must be less than or equal to K. This filter will of course reduce the size of E'_2. Thus, E_2 contains (at least) all shortest paths less than or equal to 2, assuming non-negative initial weights greater than or equal to 1. Now, let us repeat the self-join process with E_2 to generate E_3, this time preserving distances at most 4. Thus, E_3 is guaranteed to contain (at least) all shortest path lengths up to 4. If we perform $\lceil \log_2 K\rceil$ joins, the final relation will contain tuples of the form $\langle v_i, v_j, w_k\rangle$ for all vertex pairs v_i, v_j with shortest path length w_k units ($w_k \leq K$).

The algorithm described above is given explicitly in Figure 5.5. Step [1] introduces a loop that will perform all of the needed self-joins. Steps [2] – [7] perform the self-join of E_l, leaving the nonreduced result in E'_{l+1}. Steps [8] – [10] compute the appropriate minimum distances for each vertex pair, producing the reduced edge-list E_{l+1}. We call the final edge-list relation generated by the algorithm *Dist* (Step [11]). By building an index on the first column of *Dist* (Step [12]), we can use it as a lookup table to find the *K-neighborhoods* of all vertices— i.e., for a given vertex v we can quickly find the lengths of the shortest paths to all vertices that are within K units of v. Further, querying for $d(v_i, v_j)$ is also efficient: since we index *Dist*, we can access the K-neighborhood of v_i, and look for a tuple of the form $\langle v_i, v_j, w_k\rangle$. If there is such a tuple, we know the distance to be w_k. If no such tuple exists, the distance is greater than K, and we return ∞.

```
**Algorithm: Distance self-join**
**Input:** Edge set $E_1$, Maximum required distance: $K$
**Output:** Lookup table *Dist* supplies the shortest distance (up to $K$) between any pair of objects
[1] For $l$ = 1 to $\lceil \log_2 K \rceil$
[2]     Copy $E_l$ into $E'_{l+1}$.
[3]     Sort $E_l$ on first vertex. // To improve performance
[4]     Scan sorted $E_l$:
[5]         For each $\langle v_i, v_j, w_k \rangle$ and $\langle v_i, v'_j, w'_k \rangle$ in $E_l$ where $v_j \neq v'_j$
[6]             If $(w_k + w'_k \leq K)$
[7]                 Add $\langle v_j, v'_j, w_k + w'_k \rangle$ and $\langle v'_j, v_j, w_k + w'_k \rangle$ to $E'_{l+1}$.
[8]     Sort $E'_{l+1}$ on first vertex, and store in $E_{l+1}$.
[9]     Scan sorted $E_{l+1}$:
[10]        Remove tuple $\langle u, v, w \rangle$, if there exists another tuple $\langle u, v, w' \rangle$, with $w > w'$.
[11] Let *Dist* be the final $E_{l+1}$.
[12] Build index on first vertex in *Dist*.
```

Figure 5.5: "Self-Join" distance precomputation

The algorithm in Figure 5.5 runs with little I/O overhead, since sorting the data enables sequential rather than random accesses. Note that other efficient techniques are possible for computing the self-join (such as hash joins), and in fact given E_l we can use standard SQL to generate E_{l+1}. First, assume that all tables have three columns: `oid1`, `oid2`, and `dist`. Then, the following code shows how to go from E_1 to E_2; it could be parameterized and embedded within the outer loop (Step [1] of Figure 5.5) to compute the entire *Dist* table.

```
Insert into E_2
Select new_oid1, new_oid2, min(new_dist)
From
    (Select t1.oid2 as new_oid1, t2.oid2 as new_oid2, (t1.dist+t2.dist) as new_dist
    From E_1 t1, E_1 t2
    Where (t1.oid1 = t2.oid1) and (t1.dist+t2.dist <= K) and (t1.oid2 <> t2.oid2)
Union
    Select oid1 as new_oid1, oid2 as new_oid2, dist as new_dist
    From E_1)
Group by new_oid1, new_oid2
```

Unfortunately, the construction of *Dist* can be expensive. In Steps [5] – [7] of Figure 5.5, we produce the cross-product of each vertex neighborhood with itself. The size the cross-product

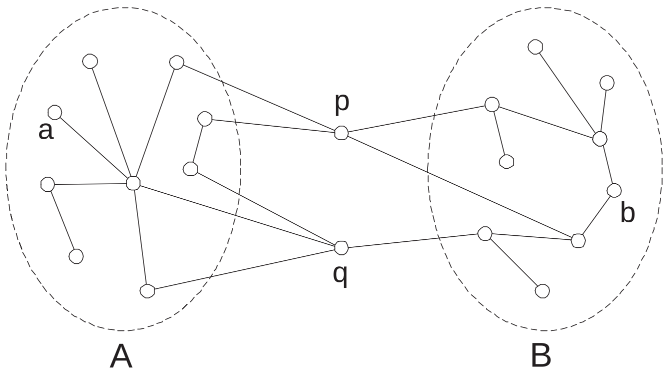


Figure 5.6: Hub vertices

could be as large as $|V|^2$ in the worst-case. For instance, when we executed the self-join algorithm on the 4MB edge-list for the IMDB database described in Section 5.1 with $K = 8$, the edge-list grew to about one gigabyte— 250 times larger than the initial input! Sorting and scanning the large nonreduced edge-lists could be expensive as well. (We have no guarantees that a SQL implementation of this algorithm will be any less expensive.) In the next section, we propose a new technique to alleviate this problem.

5.4.3 Hub Indexing

We now propose *hub indexing*, which allows us to compute shortest distances using far less space than required by the self-join algorithm of Section 5.4.2, with little sacrifce in access time. We use Figure 5.6 to explain what *hubs* are and how they can be used to compute distances efficiently. If we execute our simple self-join algorithm from the previous section on the given graph, we will explicitly store distances for all pairs of vertices in the graph. In Figure 5.6 we see that if we remove p and q, the graph is disconnected into two sub-graphs A and B. [2] Rather than storing all $|A| \times |B|$ distances, suppose we store only the $|A|+|B|$ shortest distances to p, the $|A|+|B|$ shortest distances to q, and the shortest distance between p and q. Of course, the query procedure for such an approach is slightly more complex. We can see that the shortest-path between $a \in A$ and $b \in B$ can be one of $a \sim p \sim b$ (not through q), $a \sim q \sim b$ (not through p), $a \sim p \sim q \sim b$, or $a \sim q \sim p \sim b$. We can compute $d(a, b)$ by finding these four distances and choosing the smallest.

The above description gives the reader a rough idea of our approach. By finding hubs such as p and q, we can reduce sharply the storage required for a distance index, and we will show how to handle the more complex query procedure efficiently. In addition, we can choose a hub set so that

[2] $\{p, q\}$ is known as a *separator* in graph theory.

the shortest distances between them can be stored in main memory. As we will see in Section 5.5, as we allocate more memory for hub storage, our total index size shrinks. Effectively choosing hubs in an arbitrary graph is a challenging problem, and is not the subject of this thesis. However, good hub selection algorithms based on *balanced separators* [LR88] do exist and are discussed in [GSVGM98]. Assuming we have a set of hubs, the following sections describe how to build a hub index and then answer distance queries using it.

Constructing Hub Indexes

As suggested by the above discussion, a *hub index* is represented by two key components: a *hub set* $H \subseteq V$ and a table of distances between pairs of objects whose shortest paths do not cross through elements of H. We redefine the *Dist* lookup table from Section 5.4.2 to be this new table. As we will discuss shortly, we actually transform H into a square matrix of inter-hub distances to increase the speed of the overall hub index.

Given H, we can reuse the algorithm of Figure 5.5 almost verbatim to construct the new *Dist* table. The only required change is to Step [6], which we replace with:

$$[6'] \quad \text{If } (w_k + w'_k \leq K) \text{ and } v_i \notin H$$

By checking that v_i is not in H we make sure that we do not consider any paths that cross hubs. (Paths with hubs as endpoints are still considered.) For each $v \in V$, *Dist* stores all vertices reachable within a distance of K without crossing any hubs; we call this set of vertices the "hub-bordered" neighborhood of v.

As we will explain in the next section, pairwise distances between hubs must be consulted many times to evaluate a distance query. Fortunately, experiments discussed in Section 5.5 show that even a small set of hubs greatly reduces total index size. Hence, our query algorithm assumes that the pairwise distances of all hubs are available in main memory. At index creation time, we create a square, in-memory adjacency matrix *Hubs* such that *Hubs*$[h_i][h_j]$ gives the shortest distance between hubs h_i and h_j. To do so, we first initialize each entry of *Hubs* to ∞. Then, with one sequential scan of *Dist*, for each edge $\langle h_i, h_j, w_k \rangle$, where $h_i, h_j \in H$, we set *Hubs*$[h_i][h_j] = w_k$. This step "short-cuts" the need to recompute all distances from scratch. Finally, we use Floyd-Warshall's algorithm to compute all-pairs shortest distances in *Hubs*. We must conceptually consider paths through non-hubs, but these were already accounted for when generating *Dist* tuples for paths from one hub to another. Floyd-Warshall works in-place, without requiring additional memory. Because

```
**Algorithm: Pairwise distance querying**
Input:  Lookup table on disk: Dist, Lookup matrix in memory: Hubs,
        Maximum required distance: $K$, Hub set: $H$
        Vertices to compute distance between: $u, v$ $(u \neq v)$
Return Value: Distance between $u$ and $v$: $d$
[1] If $u, v \in H$, return $d =$Hubs$[u][v]$.
[2] $d = \infty$
[3] If $u \in H$
[4]     For each $\langle v, v_i, w_k \rangle$ in Dist
[5]         If $v_i \in H$    // Path $u \sim v_i \sim v$
[6]             $d = \min(d, w_k +$Hubs$[v_i][u])$
[7]     If $d > K$, return $d = \infty$, else return $d$.
[8] Steps [4] – [7] are symmetric steps if $v \in H$, and $u \notin H$.
[9] // Neither $u$ nor $v$ is in $H$
[10] Cache in main-memory ($E_u$) all $\langle u, v_i, w_k \rangle$ from Dist
[11] For each $\langle v, v'_i, w'_k \rangle$ in Dist
[12]  If $(v'_i = u)$
[13]      $d = \min(d, w'_k)$    // Path $u \sim v$ without crossing hubs
[14]  For each edge $\langle u, v_i, w_k \rangle$ in $E_u$
[15]      If $v'_i \in H$ and $v_i \in H$    // Path $u \sim v_i \sim v'_i \sim v$ through hub vertices
[16]          $d = \min(d, w_k + w'_k +$Hubs$[v'_i][v_i])$
[17] If $d > K$, return $d = \infty$, else return $d$.
```

Figure 5.7: Pairwise distance querying

this matrix needs only to be created once, we fully materialize *Hubs* at index creation time; therefore, it is actually *Hubs* and *Dist* that comprise the hub index on disk.

Since we keep hubs and their distances in memory, a hub index has the nice property that answering a distance query requires less work on disk as more memory is made available. In fact, if the entire adjacency matrix fits in memory, we can choose H to be V and eliminate query-time disk access entirely. Our approach reveals a smooth transition to Floyd-Warshall's algorithm as main memory increases. The Proximity Engine administrator (anyone in charge of building a hub index) can specify a limit for the number of hub points based on available memory.

Querying Hub Indexes

Given the disk-based *Dist* table and the in-memory matrix *Hubs*, we can compute the distance between any two objects u and v using the algorithm in Figure 5.7. The algorithm performs a case-by-case analysis. To help explain the algorithm, we refer back to the graph in Figure 5.6, assuming $H = \{p, q\}$. Steps [1] through [8] are straightforward, since these steps handle the case where one or both of u and v are in H. (In terms of Figure 5.6, suppose that u and/or v are in $\{p, q\}$.) Steps [10] through [17] address the case where neither input vertex is in H. Steps [12] – [13] consider the case where the shortest path from u to v does not go through any of the vertices in H and its distance is therefore explicitly stored in *Dist*. (In Figure 5.6, consider the case where both vertices are in A.) Steps [14] – [16] handle shortest paths through vertices in H, such as a path from any $a \in A$ to any $b \in B$ in the figure.

If both u and v are in H, no disk I/O is performed. Recall that *Dist* is indexed based on the first vertex of each edge. Hence, if either u or v is in H, one random disk seek [3] is performed to access the hub-bordered neighborhood of v or u (Steps [4] – [8]). If neither u nor v is in H, two random disk seeks are performed to access the hub-bordered neighborhoods of both u and v (Steps [10] and [11]). The algorithm implicitly assumes that the hub-bordered neighborhood for any given vertex can be cached into memory (Step [10]). Since we use hubs, and given that K is generally small, we expect this assumption to be safe. Additional buffering techniques can be employed if needed.

Generalizing to Set Queries

The previous section discussed how to use a hub index to look up the distance between a single pair of objects. As described in Section 5.2.1, however, a *FindNear* query needs the distance between each *Find* and each *Near* object. For instance, we may need to look up the pairwise distances between $Find = \{v_1, v_2\}$ and $Near = \{v_3, v_4, v_5\}$ The naive approach is to check the hub index for each of $\{v_1, v_3\}$, $\{v_1, v_4\}$, $\{v_1, v_5\}$, $\{v_2, v_3\}$, and so on. When we have F *Find* objects and N *Near* objects, this approach will require about $2 \times F \times N$ disk seeks, impractical if F and N are large. If the *Dist* table data for all of either the *Find* or the *Near* objects fits in main memory, we can easily perform all *FindNear* distance lookups in $F + N$ seeks. If not, we can still buffer large portions of data in memory to improve performance.

In some cases, even $F + N$ seeks may still be too slow. Our movie database, for example, contains about 6500 actors. Hence, finding the result to a query like "Find actor Near Travolta" will

[3] For clarity of exposition, we do not mention any additional seeks required to navigate the index. "One" seek may translate to two or three, depending on the index.

take at least 6500 seeks. To avoid such cases, we allow engine administrators to specify object-clustering rules. For example, by clustering all "actors" together in *Dist* we avoid random seeks and execute the queries efficiently. Our engine is general enough to cluster data arbitrarily based on user specifcations. In our Lore implementation (Section 5.3), we cluster based on labels, such as "Actor," "Movie," "Producer," etc. Note that this approach increases the space requirements of *Dist*, because these clusters need not be disjoint. To mitigate the replication, preliminary investigation suggests that we can signifcantly compress vertex neighborhoods on disk, discussed further in the next section.

5.5 Performance Experiments

We now study some performance-related aspects of building hub indexes. Questions we address in this section include:

1. Given a small, fxed number of hubs, what are the space and time requirements of hub index construction?
2. How do the algorithms scale with larger databases?
3. What is the impact of selecting fewer or more hubs on the index construction time?
4. How fast is query execution? For our experiments, we used a Sun SPARC/Ultra II (2×200 MHz) running SunOS 5.6, with 256 MBs of RAM, and 18 GBs of local disk space.

To answer the above questions in a realistic scenario, we experimented with the IMDB database introduced in Section 5.1. Since the database is relatively small (the IMDB edge-list is about 4MB), we built a generator that takes an input edge-list and scales the database by any given factor S. We do not simply copy the database to scale it; rather we compute statistics on the small database and produce a new, larger database with similar characteristics. For instance, the percentage of popular actors will be maintained in the scaled-up version, and this set of actors will be acting in a scaled-up number of new movies. Similarly, movies will have the same distribution of actors from the common pool of S times as many actors, and the ratio of "romance" movies to "action" movies will stay about the same. Since our generator produces the graphs based on a real database, we believe it gives us a good testbed to evaluate our algorithms empirically. While we think the structure of our data is typical of many databases, of course it does not reflect every possible input graph.

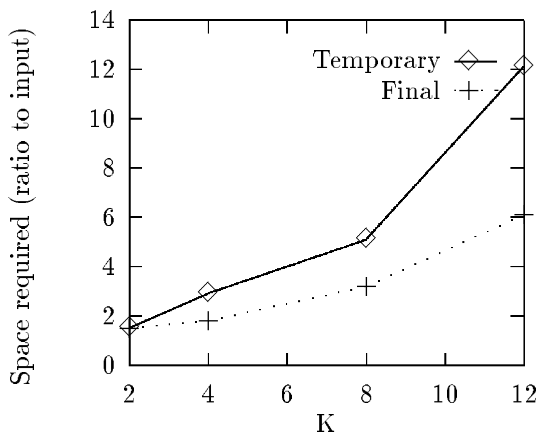


Figure 5.8: Space required with varying K

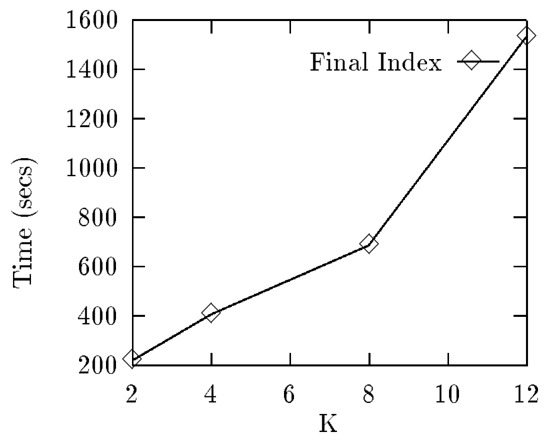


Figure 5.9: Index creation time with varying K

First, we discuss index performance when the number of hubs is fixed at a " small" number. Recall from Section 5.4.3 that the algorithm requires temporary storage (for the nonreduced edge-lists) before creating and indexing the final reduced edge-list *Dist*. For our experiments, we build an ISAM index over the final edge-list; other indexing techniques, such as a B-tree or a disk-based hash table, are of course possible. Figure 5.8 shows the temporary and final space requirements of the *Dist* table for different values of K. We define the space required as a multiple of the size of the original input. For this graph, we set scaling factor $S = 10$ and we choose no more than 2.5% of the vertices as hubs. For this case (about 40MB of data), we required less than 250K of main memory to store our *Hubs* matrix. We see that both the temporary and final space requirements for $Dist$ can get large. For $K = 12$ (the K used for our prototype in Section 5.3), the temporary and final space requirements are about 12 times and 6 times larger than the input edge-list, respectively. Similarly, Figure 5.9 reports the total time to create a hub index for different values of K. We see quadratic growth of both space and time requirements, due to the quadratic growth in the size of a vertex neighborhood. Momentarily we will show that increasing the number of hubs reduces space and time requirements.

Next, we consider how our algorithms scale as the databases grow in size. In Figure 5.10 we show the total storage required to store the final index when we (again) choose no more than 2.5% of vertices as hubs, for $K = 12$. The key point to note from this graph is that the storage consumption scales linearly, despite the fact that the large scaled databases are tightly interconnected (i.e., have many edges).

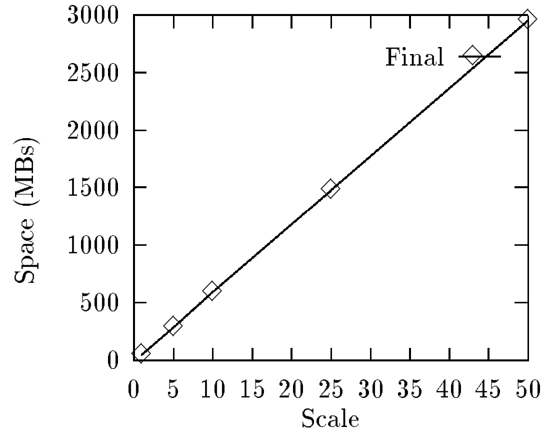


Figure 5.10: Total storage with varying scale

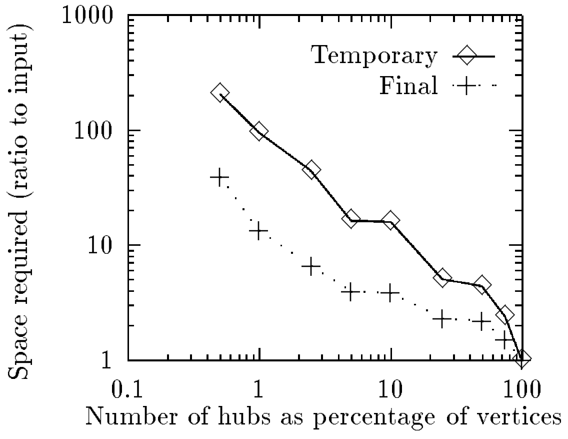


Figure 5.11: Space ratio vs. number of hubs

In Figure 5.11, we see that relatively small increases in the number of hubs can dramatically reduce the storage requirements of the *Dist* table. Again, we consider the case where $S = 10$ and $K = 12$. First, notice that if we choose fewer than 0.5% of vertices as hubs, we need significantly more space to store the *Dist* table; recall that we degenerate to the self-join algorithm when no hubs are selected. If we can choose up to 5% of vertices as hubs we see that the storage ratio for the final *Dist* table drops to about 3.93. As we mentioned earlier, the graph shows that our algorithm smoothly transitions into a main-memory shortest-path computation as more memory is made available. The index construction time also follows a trend similar to the space requirements, as shown in Figure 5.12.

Finally, we give some examples of query execution time. As can be expected, query times vary based on the size of the input sets. Consider yet again the query " Find movie Near Travolta Cage." In our (unscaled) IMDB database, $|Find| \approx 2000$ and $|Near| = 2$. With " movie" objects clustered together and no more than 2.5% of the vertices as hubs, the query takes 1.52 seconds (beyond the *FindNear* queries executed by Lore). For the query " Find movie Near location," where $|Find| \approx 2000$ and $|Near| \approx 200$, execution takes 2.78 seconds. To measure the impact of hubs on query time, first we set all vertices to be hubs and recorded the average query time q for several " representative" queries such as those proposed in Section 5.1. Then, we varied the number of hubs and issued the same queries, normalizing the query time by q. As seen in Figure 5.13, query time improves as more hubs are selected; coupled with the large decrease in index size and creation time, hub indexing is a promising technique.

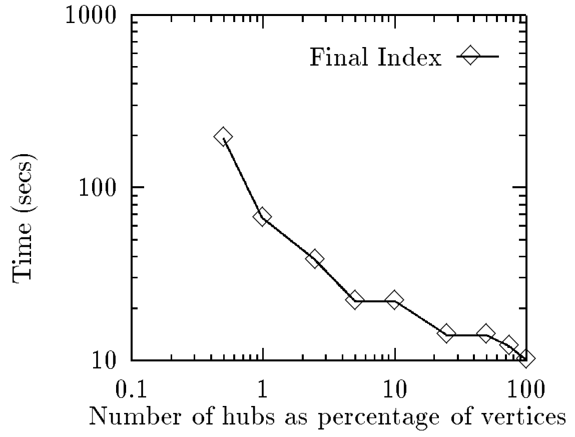


Figure 5.12: Index creation time vs. number of hubs

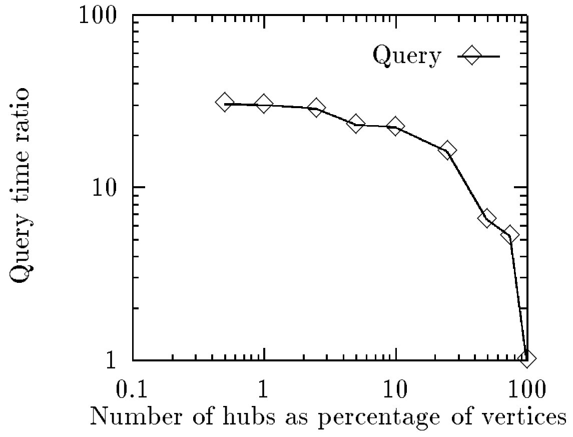


Figure 5.13: Query time vs. number of hubs

5.6 Related Work

Most existing approaches for supporting proximity search in databases are restricted to searching only within specific fields in relational databases known to store unstructured text [Ora99, DM97]. Searches do not consider interrelationships between different fields, unless manually specified through a query. One company, Data Technologies Ltd. (www.dtl.co.il), marketed technology for plain language search over databases, but to the best of our knowledge their algorithms have not been made public.

There has been extensive work on the problem of computing the *transitive closure* of a disk-resident directed graph, strictly more general than the problem of computing shortest distances up to some K. [DR94] examines many algorithms for this problem and supplies comparative performance evaluation, as well as discussion of useful measures of performance. In principle, it is possible to apply these algorithms to our problem. However, full transitive closure is a somewhat more general problem from our shortest-paths problem, so our specialized algorithms perform better. Furthermore, the algorithms in [DR94] perform all computations at query time.

Chapter 6

XML Support in Lore, DataGuides, and Proximity Search

The recent emergence of *XML* (the *eXtensible Markup Language*) as a new standard for data representation and exchange on the World-Wide Web has drawn significant attention [XML98]. As discussed in Chapter 1, there is a striking similarity between XML and semistructured data models such as OEM (Chapter 2). Until now, this thesis has focused on OEM, which formed the original basis for most of our research. This chapter describes the modifications and extensions we have made to apply our work to XML as well. In Section 6.1, we provide some background on XML. Next, in Section 6.2, we define an XML data model— a subtly challenging task given that XML itself is just a textual language. In Section 6.3 we give an overview of how we can encode our XML data model in OEM. We describe briefly changes to *Lorel,* Lore's query language, in Section 6.4. Next, we focus on how we can change DataGuides (Chapter 3) and proximity search (Chapter 5) to work with XML, in Sections 6.5 and 6.6, respectively. We pay special attention to how we can incorporate the inherent ordering present in XML data, since our original work was in the context of the unordered OEM model.

Some of the work in this chapter first appeared in [GMW99].

6.1 XML Background and Comparison With OEM

XML is a textual language quickly gaining in popularity for data representation and exchange on the Web [XML98]. XML data is specified in documents, containing nested, tagged *elements*. Lexically, each tagged element has a sequence of zero or more attribute/value pairs, and a sequence of zero or more *subelements*. These subelements may themselves be tagged elements, or they may be "tagless" segments of text data. Consider the following simple example.

```
<DBGroup>
    <Member Name="Smith">
        <Age>28</Age>
    </Member>
    <Member>
        <Name>Jones</Name>
        <Advisor>Ullman</Advisor>
    </Member>
    <Project>
        <Title>Lore</Title>
    </Project>
</DBGroup>
```

Note that XML poses no restrictions on consistency across tags: `Name` is an attribute of one `Member` element, and it is a subelement of the other.

Because XML was defined as a textual language rather than a data model, an XML document always has implicit order— an order that may or may not be relevant but is nonetheless unavoidable in a textual representation. A *well-formed* XML document places no restrictions on tags, attribute names, or nesting patterns. Alternatively, a document can be accompanied by a *Document Type Definition (DTD)*, essentially a grammar for restricting the tags and structure of a document. An XML document satisfying a DTD grammar is considered *valid*. While not exactly a data model, a standard *Document Object Model* (*DOM*) for XML has been defined [ABea98], allowing XML to be manipulated by software. The DOM defines how to translate an XML document into data structures and thus can serve as a starting point for any XML data model.

In contrast, recall that OEM (Chapter 2) is not an ordered data model: each object has an unordered set of subobjects. Further, OEM does not have an analogous concept of attributes— only subobjects. More subtly, in OEM labels on edges are used only as entry points (Section 2.2 and to denote relationships— an OEM object need not have a single label that it "owns." In contrast, the XML DOM specifies that each (non-text) element contains its own identifying tag. Another difference is that the XML DOM today does not support graph structure directly, no doubt an artifact of XML's document orientation. Currently, XML uses special attribute types to encode graph structure. An element can have a single attribute of type *ID* (as specifed in the DTD) whose value provides a unique identifier that can be referenced by attributes of type *IDREF* or *IDREFS* from other elements. Let us extend the above example to have such attributes:

```
<DBGroup>
   <Member Name="Smith" Advisor="m1">
      <Age>28</Age>
   </Member>
   <Member ID="m1" Project="p1">
      <Name>Jones</Name>
      <Advisor>Ullman</Advisor>
   </Member>
   <Project ID="p1" Member="m1">
      <Title>Lore</Title>
   </Project>
<DBGroup>
```

Assume attribute ID is of type ID, and that attributes Project, Advisor, and Member are of type IDREF. The above example encodes a graph where the Member, Project, and Advisor attributes serve as labeled references to the elements with corresponding ID attributes.[1]

XML's "second-class" support of graph structure leads to interesting decisions in specifying a true data model and query language. Should an XML data model be a tree that corresponds to XML's text representation (like the DOM), or a graph that includes the intended links? Our view is that both approaches are important. In some situations, an application may wish to process XML data as a literal tree, where IDREF(S) attributes are nothing more than text strings. In other situations, an application may wish to process XML data as its intended semantic graph. Our decision is to support both modes— *literal* and *semantic*— that a user or application can select between. The choice of mode has a direct impact on query evaluation and results, as we will see later.

6.2 Lore's XML Data Model

In our data model, an XML element is modeled as a pair $\langle eid, value\rangle$, where *eid* is a unique *element identifier*, and *value* is either an atomic text string or a complex value containing the following four components:

[1] Unfortunately, as of this writing a DTD is required to specify attribute types, so today it is common to use inelegant heuristics to deduce ID/IDREF/IDREFS types when a DTD is not available. We assume that any attribute named ID is of type ID, and that any attribute whose value is appears elsewhere as an ID value is of type IDREF. If the attribute is a sequence of space-separated ID values, we assume type IDREFS.

1. A string-valued tag corresponding to the XML tag for that element.
2. An ordered list of attribute-name/atomic-value pairs, where each attribute-name is a string and each atomic-value has an atomic type [2] drawn from integer, real, string, etc., or ID, IDREF, or IDREFS.
3. An ordered list of *crosslink subelements* of the form ⟨*label*, *eid*⟩, where *label* is a string, introduced via an attribute of type IDREF or IDREFS.
4. An ordered list of *normal subelements* of the form ⟨*label*, *eid*⟩, introduced via lexical nesting within an XML document.

We differentiate normal subelements (4) from crosslink subelements (3) so we can support both literal and semantic modes, as motivated in Section 6.1.

An XML document is mapped easily into our data model. Note that we ignore comments and whitespace between tagged elements. As a base case, text between tags is translated into an atomic text element; we do the same thing for CDATA sections, used in XML to escape text that might otherwise be interpreted as markup [XML98]. Otherwise, a document element is translated into a complex data element such that:

1. The tag of the data element is the tag of the document element.
2. The list of attribute-name/atomic-value pairs in the data element is derived directly from the document element's attribute list.
3. For each attribute value i of type IDREF in the document element, or component i of an attribute value of type IDREFS, there is one crosslink subelement ⟨*label*, *eid*⟩ in the data element, where *label* is the corresponding attribute name and *eid* identifies the unique data element whose *ID* attribute value matches i.
4. The subelements of the document element appear, in order, as the normal subelements of the data element. The label for each data subelement is the tag of that document subelement, or *Text* if the document subelement is atomic.

Once an XML document is mapped into our data model it is convenient to visualize the data as a directed, labeled, ordered graph. The nodes in the graph represent the data elements and the edges

[2] While the XML specification does not include attribute types, some extensions do and we have chosen to include them.

```
<DBGroup>
  <Member Name="Smith" Advisor="m1">
    <Age>28</Age>
  </Member>
  <Member ID="m1" Project="p1">
    <Name>Jones</Name>
    <Advisor>Ullman</Advisor>
  </Member>
  <Project ID="p1" Member="m1">
    <Title>Lore</Title>
  </Project>
</DBGroup>
```

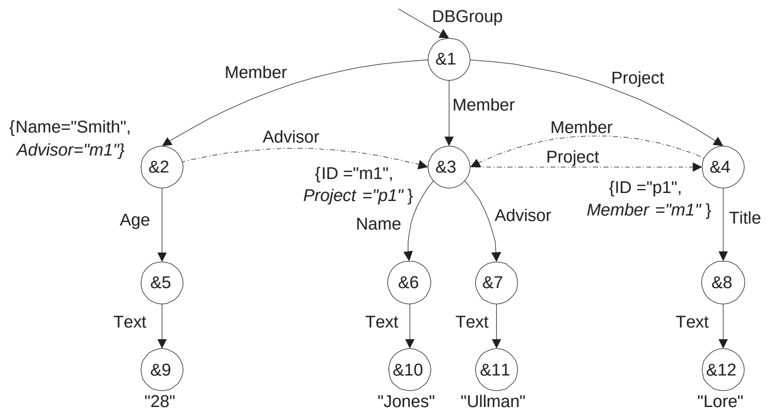


Figure 6.1: An XML document and its graph

represent the element-subelement relationship. Each node representing a complex data element contains a tag and an ordered list of attribute-name/atomic-value pairs; atomic data element nodes contain string values. There are two different types of edges in the graph: (i) normal subelement edges, labeled with the tag of the destination subelement; (ii) crosslink edges, labeled with the attribute name that introduced the crosslink. Notice that the graph representation is isomorphic to the data model, so they can be discussed interchangeably.

As mentioned earlier, it is useful to view the XML data in one of two modes: *semantic* or *literal*. Semantic mode is used when the user or application wishes to view the database as an interconnected graph. The graph representing the semantic mode omits attributes of type IDREF and IDREFS, and the distinction between subelement and crosslink edges is gone. Literal mode is available when the user wishes to view the database as an XML document. IDREF and IDREFS attributes are visible as textual strings, while crosslink edges are invisible. In literal mode, the database is always a tree.

Figure 6.1 shows the small sample XML document from Section 6.1 and the graph representation in our data model. Eids appear within nodes and are written as &1, &2, etc. Attribute-name/atomic-value pairs are shown next to the associated nodes (surrounded by {}), with IDREF attributes in italics. Subelement edges are solid and crosslink edges are dashed. The order between subelements is left-to-right. We have not shown the tag associated with each element since it is straightforward to deduce for this simple database. (For example, node &3 has the tag *Member* and not *Advisor*.) Note that the root tag of the XML document is modeled as an object with an incoming label, in the spirit of *entry points* as introduced in Section 2.2. In semantic mode, the database in Figure 6.1 does not include the (italicized) IDREF attributes. In literal mode, the (dashed) crosslinks are not included.

6.3 Encoding XML in OEM

In Lore, we decided to encode our XML data model in OEM. Because the graph-based XML model defined in Section 6.2 is so similar to OEM, the mapping is quite straightforward. Further, an encoding strategy allowed us to reuse many of the same algorithms and much of the same code in Lore for XML. In particular, as we will discuss in Sections 6.5 and 6.6, we were able reuse our existing approaches for computing DataGuides and proximity search over OEM for XML as well, with only a few changes discussed in those sections.

The graph presented in Figure 6.1 is " almost" OEM; it represents much of our encoding scheme. In particular, elements correspond to objects, and incoming object edges are labeled with either the tag of the destination subelement (for normal subelements) or, for crosslink edges, the attribute name that introduced the crosslink. (In literal mode, crosslink edges are left out entirely.)

We assume that an XML graph is ordered. While OEM itself is unordered, we assume that a system supporting OEM (such as Lore) can indeed preserve the order of data as it is loaded. Thus, when XML is encoded in OEM within Lore, we create an *ordered* OEM graph. This order is preserved during query processing, assuring that an XML query will return data in the same order it was created.

There are two important differences in our actual OEM encoding. First, each non-crosslink attribute A of an element E is encoded as an atomic subobject O_A of the object corresponding to E: the label A is marked specially so that the system knows the subobject represents an attribute and not a subelement, and the value of attribute A is stored as the value of O_A. Second, every object can, when necessary, store special " metadata" that contains its true XML tag, along with information that differentiates between normal subelements and crosslinks. Intentionally, most of the Lore system does not need to be aware of this special metadata.

Note that Lore has no problem processing and displaying raw OEM as XML. Subobjects are always translated to subelements, since OEM has no notion of attributes. Atomic values are translated to free text. Further, since there is no special " metadata," tag names are always derived from the incoming edge label traversed to display the element, and there are no crosslinks. Finally, several different techniques are available for breaking OEM cycles for the purposes of serial XML output.

Qualification	Symbol	Example	Semantic Matches	Literal Matches
Subelements only	>	DB.Member.>Name	&6	&6
		DB.Member.>Advisor	&3, &7	&7
Attributes only	@	DB.Member.@Name	"Smith"	"Smith"
		DB.Member.@Advisor	*empty*	"m1"
None	None	DB.Member.Advisor	&3, &7	&7, "m1"

Table 6.1: Path expression qualifiers

6.4 Lore's XML Query Language

We now briefly discuss modifications we have made to the Lorel query language (Chapter 2) to accommodate the differences between our new XML data model and OEM, and to exploit XML features not present in OEM. Although not a core contribution of this thesis, this work was a natural extension of our efforts to make the Lore system support XML "across the board." Recall that a database in our XML data model can be interpreted either in *semantic mode* or in *literal mode*. For simplicity let us assume that the desired mode is selected for each query posed against the database.

Distinguishing between attributes and subelements. Recall from Chapter 2 that *path expressions* are the basic building blocks of Lorel, and that during query evaluation, path expressions are matched to paths in the database graph. For XML, we extend the meaning of path expressions to navigate both attributes and subelements, and we introduce *path expression qualifiers* in order to distinguish between the two when desired. We use the optional symbol > before a label to indicate matching subelements only, and the optional symbol @ to indicate matching attributes only. When no qualifier is given, both attributes and subelements are matched— we expect this to be the most common case. Table 6.1 shows simple examples of path expressions with qualifiers applied over the database in Figure 6.1. Recall from Section 6.2 that in semantic mode IDREF(S) attributes are not visible, while in literal mode IDREF(S) are treated like other attributes and crosslink edges are not visible.

Comparisons. We anticipate that many different kinds of comparisons may be useful in queries over XML data. For example, constants might be compared against attribute values or against element text. In other settings, we might want to compare against a serialization of all text elements in an XML subtree, ignoring markup. In graph-structured data, we might want to test for eid equality.

Rather than support many distinct comparison operators, we decided instead that for the purpose of comparisons we would treat each XML component as some kind of atomic value, either through default behavior or via explicit transformation functions.

Attribute values are always atomic. For elements, Table 6.2 describes several built-in functions that can be used to transform an element into a string, and can be used outside of comparisons if desired, e.g., in the `select` clause. (Each function returns NULL if called over an attribute instead of an element.) Since it is inconvenient for a user to have to specify functions for every comparison, keeping in the spirit of Lorel we set default semantics when functions are not supplied based on our impression of the most common and intuitive uses:

1. For an atomic (`Text`) element, the default value is the text itself.

2. For elements that have no attributes and only one or more `Text` elements as children, the default value is the concatenation of the children's text values (a restricted case of the *Concatenate* function).

3. For all other elements, the default value is the element's eid represented as a string (the *Eid* function).

Example: Suppose we are looking for a group member whose advisor is " Ullman" . In the original version of Lorel, `DBGroup.Member.Advisor=" Ullman"` does the trick. Based on Figure 6.1 it appears that for our XML data model we must write `DBGroup.Member.Advisor.Text=" Ullman"` , and indeed this expression will give us the correct answer. However, the former comparison also will give us the correct answer by virtue of default semantics case (2) above. In general, we have found that most Lorel queries designed for an OEM database can be used unmodified on a corresponding XML database, such as the simple example we have just shown.

Additional features. Lorel has been extended with several additional features related to XML, described in [GMW99]. These features include *range qualifers* for requesting specifc ranges of subelements based on the order of subelements with a given tag; a way to order query results based on order in the original document; transformation and restructuring of query results; and modifcations to the Lorel update language.

Function	Description
Flatten(e)	Ignoring all tags, recursively serializes all text values in the subtree rooted at element e (following normal subelements only).
Concatenate(e)	Concatenates all immediate text children of an element and ignores all other subelements.
Tag(e)	Returns the XML tag of an element.
Eid(e)	Returns a string representation of the eid of element e.
XML(e)	Transforms the graph, starting with element e, into an XML document. Note that there is no single " correct" way to generate an XML document from graph-structured data, so it will be difficult to use this option to compare against string constants.

Table 6.2: Functions to produce different interpretations for comparisons

6.5 DataGuides for XML

As described in Chapter 3, a *DataGuide* is a concise, accurate structural summary of a semistructured database. DataGuides are constructed and maintained dynamically from a database, and they have proved useful for a variety of purposes: browsing, query formulation, storing statistics, query optimization, and most recently compression of XML data [LS00]. DataGuides were defined in Chapter 3 in the context of the OEM model: DataGuides summarize unordered OEM databases, and a DataGuide is itself an unordered OEM object. Recall from Chapter 3 that a DataGuide G of a graph-structured source database D is itself a graph such that every label path from the root of D appears exactly once in G, and every label path from the root of G appears in D.

It is straightforward to use our original DataGuide algorithms to create and maintain unordered XML DataGuides over XML data. We take advantage of the fact that the XML is encoded in OEM, as discussed in Section 6.3. First, we decide whether the OEM encoding should reflect literal mode (in which the case the OEM database is a tree) or semantic node (in which the database is a graph). In either case, we run the original DataGuide algorithm as-is over our OEM encoding, and the resulting structure can be interpreted (exported) as well-formed XML. Because atomic values are by definition left out of DataGuides, attributes in an XML DataGuide are empty (e.g., NAME=" "), and any free text between tags is omitted as well. Alternatively, the XML DataGuide can be explored as graph in the same way we explore OEM DataGuides (Chapter 3)— with the small difference that attributes are marked with a @ and other subelements are marked with a $>$. Currently the special " metadata" mentioned in Section 6.3 is ignored during DataGuide creation. Tree structured data is

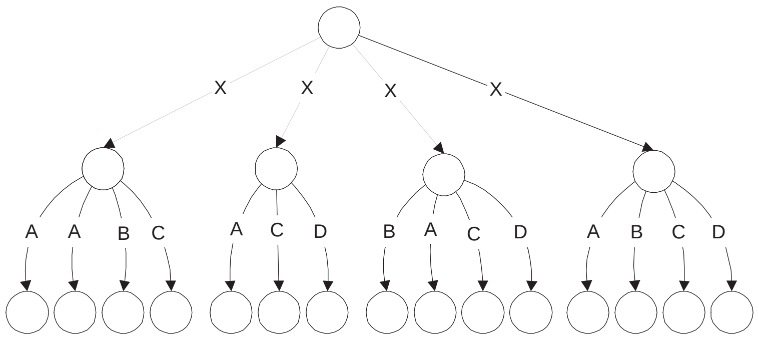


Figure 6.2: Graph representation of XML for DataGuide example

not affected, but for graph-structured data the notion of an element's " primary" tag is lost; instead, tag names are derived based on incoming edges during traversal as described in Section 6.3, and sublements links are not distinguished from crosslinks.

Before further discussion of XML DataGuides, let us address briefly the relationship between XML DTDs and DataGuides. Recall from Section 6.1 that a DTD is a grammar that restricts the tags, attributes, and nesting structure of an XML document. DTDs are not required to accompany XML; when a DTD is not supplied, the notion of a DataGuide is just as important for XML as for OEM. When a DTD is available, we can build an *Approximate DataGuide* (see Section 3.7) from the DTD that can be used by the Lore system, such as in the user interface and for the query warning system (Chapter 3). Note that DTDs currently do not support graph structure beyond restricting attribute types to ID and IDREF(S), so DataGuides are more expressive than DTDs in this regard.

In the remainder of this section, we address the issue of ordering in XML DataGuides. Consider the following tiny snippet of abstract XML data, contrived to illustrate a point.

```
<X><A/><A/><B/><C/></X>
<X><A/><C/><D/></X>
<X><B/><A/><C/><D/></X>
<X><A/><B/><C/><D/></X>
```

The graph representation of this XML is shown in Figure 6.2; in this case, since there are no attributes or free text, translation to and from OEM is trivial. (The OEM graph is actually ordered on disk, as discussed in Section 6.3; however, the original DataGuide algorithm ignores order.) Using our standard DataGuide algorithm over this graph, the result is shown in Figure 6.3. The output of the original DataGuide algorithm is an unordered graph. For this example, our challenge is to impose a useful order over the subelements of X to somehow reflect the order of the original data.

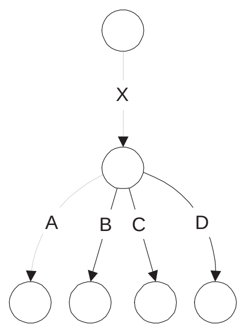


Figure 6.3: DataGuide for Figure 6.2

In particular, one possible permutation is:

<X><A/><B/><C/><D/></X>

Given the DataGuide graph, each remaining permutation of the A, B, C, and D subelements forms an equivalent representation of Figure 6.3.

When we take order into account, we would like to preserve the original definition of a DataGuide as much as possible, but extend the definition to summarize the order of subelements as well as the overall structure. We thus propose to keep the size of the ordered DataGuide the same as the size of the unordered DataGuide, choosing the " best" subelement ordering for each element in the DataGuide. (If we want to store further information about the actual orderings, we can *annotate* the DataGuide elements, as described in Section 3.2.3.)

In our example, intuitively ABCD does the best job of approximating the subelement order for the X instances in the source data: A is the first subelement in 75% of the instances; B follows A in two instances and precedes it in one; C follows A and B in all three instances where they all appear; and D is last in the three instances it's a part of. While it may be easy to choose a " best" order for this simple example, it is a challenge to define the " best" order for an XML DataGuide in general, and the definition could easily change depending on the application. Hence, we have devised several strategies for summarization and report on their effectiveness through an experimental framework.

6.5.1 Problem Formulation

The problem of ordering a DataGuide can be broken down recursively into the problem of ordering the subelements of each DataGuide element. (If we also wish to order the attributes of each element, the problem can be treated in the same way.) Suppose we create an XML DataGuide G of a source database D. Consider any element e in G, reachable from G's root by some sequence of tags p. (By the definition of a strong DataGuide, e is the only element in G reachable via p; see Section 3.2.1.) Let T be the set of elements in database D reachable by p. (In Section 3.2.1 we define T as the *target set* of p in D.) By the original DataGuide definition, each unique subelement tag of the elements in T appears exactly once as a subelement tag of e, and as discussed above we retain this requirement in the presence of order. To order the DataGuide, we must order the subelements of each such element e. We will do so based on subelement ordering in all of the elements in T.

The problem can be stated more formally (and abstractly) as follows. Consider a set $S = \{\sigma_1, \ldots, \sigma_n\}$, where each σ_i is a sequence of labels. Construct a single sequence σ of labels that "best" summarizes the sequences in S, where σ contains each label appearing in any σ_i exactly once. S corresponds to target set T, $\sigma_1, \ldots, \sigma_n$ to the elements in T, and each σ_i encodes the subelement ordering of one element of T. In our simple example at the start of this section, $S = \{$AABC, ACD, BACD, ABCD$\}$ and we constructed $\sigma =$ABCD.

6.5.2 Algorithms

We now describe three proposed algorithms for solving the problem specified in Section 6.5.1. Note that for the simple example given at the beginning of Section 6.5, all three algorithms select ABCD (which was seen to be the "best" permutation). The algorithms are evaluated experimentally in Section 6.5.3.

Greedy

One option is to use a simple greedy algorithm to generate σ from $S = \{\sigma_1, \ldots, \sigma_n\}$. To begin, select the label L that appears at the head of the largest number of sequences in S. Label L becomes the first label in σ. Remove all instances of L from S, and repeat the process until all sequences in S are empty. σ will contain all labels exactly once. This algorithm is simple and can effectively summarize sequence order in many cases, but there are several situations where it can produce counterintuitive results. Consider:

$S = \{$BABB, BABB, BABB, ABB, ABB, XABB$\}$

For this input, the greedy algorithm will construct BAX. However, this choice does a poor job of reflecting the fact that A precedes B in the data far more often than B precedes A.

Edit Distance

A more intricate algorithm can be constructed using *string edit distance* [Gus97], which measures the minimum number of character insertions, deletions, or changes required to transform one string into another. (For example, the words wall and still have an edit distance of 3: starting with wall, we can change the w to s, change the a to t, and insert an i. Note that edit distance is symmetric.)

With a brute force approach, we can consider as candidates for σ all permutations of all labels in any σ_i. Then we compute the sum of the edit distances from each candidate σ to all of the sequences in S. The σ permutation with the minimum overall edit distance is selected. For the example sequence S above, ABX and AXB tie as the best permutations according to this algorithm.

There are many possible ways to further tune this approach. For example, different costs may be assigned to different edit functions: to account for consecutive labels in an input sequence, we might want to set the cost of a label deletion to be cheaper if it's exactly the same as either adjacent label. Another possibility is to use a non-linear combination of the edit distances from σ to the sequences in S to enhance (or mitigate) the impact of any particular sequence within the set.

Unfortunately, this algorithm can be extremely expensive computationally, so pruning strategies would be essential to making it practical in general.

Weighted Averages

For our third algorithm, we calculate the average sequence position for every label across all sequences in S and then pick a final sequence that most closely matches the average sequence number for each label. Here, we explicitly collapse consecutive identical labels to compute sequence positions.

More specifically, consider $S = \{\sigma_1, \ldots, \sigma_n\}$. First, for each σ_i and each unique label L in σ_i, we compute $pos(L, \sigma_i)$: the average position of L in σ_i, after collapsing consecutive identical labels. As a simple example, pos(B, AAABBBCC) is 2, since after collapsing consecutive labels B is the second label. When a label appears in more than one position in a sequence, we use the number of consecutive instances at each position to weight the final average. For example, pos(B, BBBAB) is 1.5: $(3 \times 1 + 3)/4$, while pos(B, BAB) is 2: $(1 + 3)/2$. Finally, let S_L be the set of sequences σ_i in S such that L appears in σ_i. The final average position for L is computed as:

$$\frac{\sum_{\sigma_i \in S_L} pos(L, \sigma_i)}{|S_L|}$$

To illustrate this algorithm, consider the following input data.

$S = \{$AAABCDC, BAC, AAACCCDC$\}$

For this data, we compute the following positions:

pos(A, σ_1) = 1; pos(A, σ_2) = 2; pos(A, σ_3) = 1
pos(B, σ_1) = 2; pos(B, σ_2) = 1
pos(C, σ_1) = 4; pos(C, σ_2) = 3; pos(C, σ_3) = 2.5
pos(D, σ_1) = 4; pos(D, σ_3) = 3

The final positions for A, B, C, and D are approximately 1.3, 1.5, 3.2, and 3.5, respectively, leaving ABCD as the final choice for σ. For the example sequence S in Section 6.5.2, this algorithm selects XAB.

6.5.3 Experimental Framework and Performance Results

To evaluate our different algorithms over large data sets, we created a simple program that generates sets of label sequences with varying characteristics. The language of possible labels consists of the letters A– Z, both in upper and lower case. The program takes a lottery-based approach to picking labels to construct a sequence. Given an input integer parameter t, the first label picked will be t times more likely to be an A than any other letter (each of which has an equal chance to be picked). The second label picked will be t times more likely to be a B than any other letter, and so on, for up to l letters, where l is an input parameter that can vary from 1 to 26. The lower-case letters will be addressed momentarily. Each time a label is selected, we have another "lottery" to determine how many consecutive instances of that label to include in the sequence. We make it equally likely that 1, 2, 3, . . ., f consecutive instances are included, where f is an input parameter. A third parameter is n, for noise, intended to model the occasional inclusion of atypical labels— as may happen with semistructured data. Before selecting each new label for the sequence, with chances 1 out of n we will insert a randomly selected lower-case letter into the sequence.

As an example let us set l=5 (for 5 upper-case labels), t=20 (making the odds 20/24 that the right label will be picked at each step, since the other four labels all have an equal chance of being picked), f=5 for moderate repetition, and n=10 for some noise. The following results represent one run to generate 10 sequences.

```
cABBBBBBEEEE
cAAAABBBBADDDDBB
```

AAAAcCCCDDDE
AAAABBBBCCCCDDDEEEE
hAAABnCCDDEE
AAAAAhCCDDDDEE
AAAAAAACDDDDbEE
AEEEECCCBEEE
ABCCCCaDDE
ABBBBCCCDE

To compare the effectiveness of our algorithms from Section 6.5.2, we measure how often each algorithm chooses a "correct" permutation as we vary the input parameters. In this setting, we say a permutation is "correct" if A, B, C, and so on all appear within the permutation in lexicographic order. That is, A precedes B, which precedes C, and so on. We ignore any noise labels when determining whether a permutation is correct. A good algorithm should select a "correct" permutation more often as t increases (since labels are more likely to be chosen in order) and as n increases (since noise is less likely to be added between labels). Furthermore, if an algorithm can consistently select the correct result for smaller t and n, then it is likely to do well in practice at finding intuitive permutations.

Note that we are not measuring running time in these experiments, only effectiveness in selecting a good permutation. Both the weighted averages and greedy algorithms are linear in running time with respect to the total number of labels in all the sequences, while the edit distance algorithm shows factorial space and time growth in the number of distinct labels across sequences. The number of different "noise" labels quickly makes the edit distance algorithm infeasible, so in our experiments we make one small adjustment: we consider all permutations of the "primary" labels (A, B, C, etc.) only, then measure edit distances to the original sequences, which include both primary and noise labels.

We compare the effectiveness of our algorithms in Figure 6.4. Six graphs are presented, corresponding to six different values for t (from 2 to 7). In each graph, we show the effectiveness of all three algorithms (WA for weighted averages, ED for edit distance, and GR for greedy) for six different values of n, again from 2 to 7. For each combination of t and n, we ran all three algorithms over 20 independent sets of 100 sequences. The effectiveness is the percentage of the 20 runs that return a correct permutation. We set $l = 5$ and $f = 5$ for all experiments, though we observed similar results for other values.

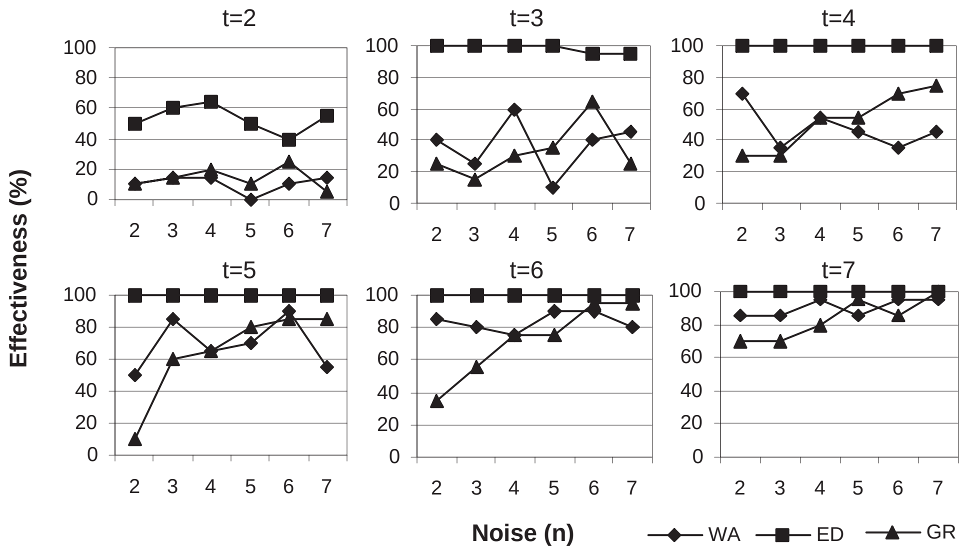


Figure 6.4: Comparison of ordered DataGuide algorithms

The results show quite conclusively that the edit distance algorithm is the most effective, reaching about 100% effectiveness for all $t \geq 3$, at any value of n. When noise is rare, the other two algorithms are similar to each other in accuracy, approaching 100% effectiveness when $t \geq 6$ and $n \geq 6$. The greedy algorithm is the most susceptible to large amounts of noise: it is less effective than the weighted average algorithm when $n \leq 3$.

6.6 Proximity Search

Proximity search is a concept from information retrieval (IR) that we applied to searching graph-structured databases, described in Chapter 5. Our proximity search technique is general: it can be used over any database that can be modeled as a graph of interconnected objects. As we described in Section 6.3, XML does have a straightforward mapping to OEM; hence, proximity search works without changes over XML data. For example, by representing element attributes as children of their parent node, our proximity search technique can identify that an element's attributes are " near" their parent. As with our DataGuide work, however, our original work on proximity search was based on an unordered model. In the remainder of this section, we show how to augment our graph representation of XML data such that shortest path computations account for subelement order. We demonstrate the impact of our changes in a sample scenario where subelement order is clearly relevant to proximity search.

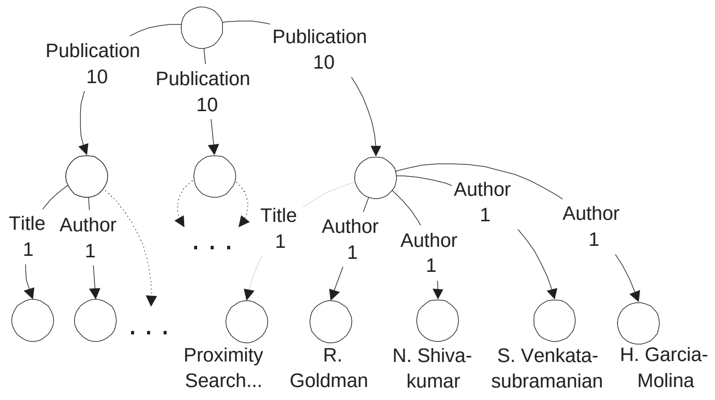


Figure 6.5: Original XML graph

Consider the following sample XML data, representing three DBGroup publications within a larger XML database.

```
<Publication>
    <Title>DataGuides: Enabling Query Formulation and Optimization in Semistructured
            Databases</Title>
    <Author>R. Goldman</Author>
    <Author>J. Widom</Author>
</Publication>
<Publication>
    <Title>Lore: A Database Management System for Semistructured Data</Title>
    <Author>J. McHugh</Author>
    <Author>S. Abiteboul</Author>
    <Author>R. Goldman</Author>
    <Author>D. Quass</Author>
    <Author>J. Widom</Author>
</Publication>
<Publication>
    <Title>Proximity Search in Databases</Title>
    <Author>R. Goldman</Author>
    <Author>N. Shivakumar</Author>
    <Author>S. Venkatasubramanian</Author>
    <Author>H. Garcia-Molina</Author>
</Publication>
```

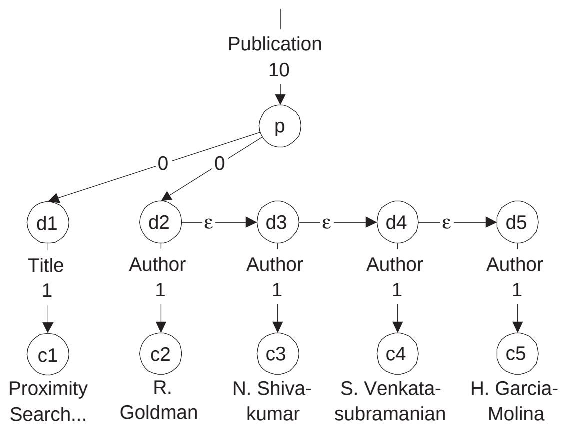


Figure 6.6: XML graph transformed for ordered proximity search

Consider a search that tries to identify publication titles "near" J. Widom. The traditional definition of textual proximity does not work well in this case: "J. Widom" is closer in the text to "Proximity Search in Databases" than to "Lore", even though she isn't an author of the former. Our initial work on proximity search addresses exactly this situation (Chapter 5). We model the data as a graph, as described in Section 6.3, and users can optionally add weights on edges to indicate the "strength" of object-subobject relationships. Distance between data objects is then measured based on the shortest weighted path in the graph, and a special index is built to speed up the computation, as described in Chapter 5. Even with uniform weights, the graph encoding and shortest path approach to proximity search solves the "Title near J. Widom" problem above.

Now let us consider the impact of order. In the example above, the order of authors is a very important aspect of the data. If we want to find "Publication near Goldman", the publications where "R. Goldman" is a first or second author should rank higher than those where "R. Goldman" is a later author. To incorporate order into our proximity search framework, we preprocess each XML data graph before building our proximity search index, adding new objects and weighted edges that adjust the weighted shortest paths between objects in order to reflect ordering. For example, Figure 6.5 shows the XML data above modeled as a graph as in Figure 6.1, with some simple weights to illustrate our approach.[3] Figure 6.6 shows the transformed graph for the rightmost publication subtree. As can be seen, the first Author is distance 1 from the parent publication element, the second author is distance $1 + \epsilon$ (for some small value ϵ), the third is distance $1 + 2\epsilon$, etc. The

[3] For improved presentation, here we simplify the XML graph slightly and omit Text subelements, instead showing text as atomic values of tag elements. Note that the transformations described do apply to our full encoding of the the XML data model, including attributes and Text elements. Further, by showing the three publications under a root object we are assuming that the XML publication elements are part of a larger XML document.

distance between the first and second author is $2 + \epsilon$, the same as the distance between the second and third author and the distance between the third and fourth.

In this case we decided that the Title subelement and the first Author should be the same distance from the parent. However, if we preferred to take into account the fact that Title precedes the first Author, we could insert an ϵ edge between nodes d_1 and d_2 and remove the edge from d_2 to p.

In the general case, consider an element p. We partition p's list of subelements into sublists, where order is relevant within but not between sublists.[4] For each subelement c_i of p, we "disconnect" c_i from p and create a new parent d_i of c_i. The weight on the $d_i \to c_i$ edge is the weight from the original $p \to c_i$ edge. We then connect adjacent d_i nodes within each sublist with edges of weight ϵ, and finally add an edge with weight 0 from p to the first d_i for each sublist. Note that it is well-defined to perform this transformation on arbitrary graphs.

The transformation effects some important properties. Consider p and one of its sublists L such that the weight from p to every element in L is W. First, we see that any two adjacent siblings are now $2W + \epsilon$ apart. More generally, the ith and jth siblings are $2W + (|i - j| \times \epsilon)$ apart, assuring that sibling distances grow monotonically as their separation within the list increases. Also, the distance from p to the ith element (counting from 0) of L is $W + (i \times \epsilon)$, assuring that distance from the parent to its children grows monotonically beginning with the first child.

6.6.1 Examples

To demonstrate the impact of our transformation, we built two proximity indexes over an XML version of the DBGroup database used throughout this thesis. One index was based on the original graph, and the other based on the transformed graph to take ordering into account. The format of the publication data is similar to the XML shown above, though there is only one XML element per unique author, referenced (via an IDREF) from all of his or her publications.

We performed several searches using both indexes and compared results. Recall that the result of a proximity search is a ranked list of *Find* elements, where the score of each is based on proximity to all elements in the *Near* set. The score also is influenced by several tuning parameters described in Chapter 5; we use the same default parameters as in Chapter 5. We describe results from two representative searches.

First, we performed the search "Find Publication near Goldman". All publications where R.

[4] This partitioning could be provided by the XML content author. As a default when not specified, we either create one sublist containing all subelements or we partition based on repeating XML tags. In our example, tags Title and Author partition the subelements into two sublists. When present, XML attributes can be grouped into their own sublist.

Goldman was the first author received the highest score, followed by publications where he was the second author, etc. In the original scheme, all of R. Goldman's publications received the same score.

Next, we performed the search "Find Author near Goldman." The XML element for R. Goldman received the highest score. Next was an eight-way tie among S. Abiteboul, S. Chawathe, A. Crespo, H. Garcia-Molina, J. McHugh, D. Quass, N. Shivakumar, and V. Vassalos. All were adjacent to R. Goldman in an author list except for A. Crespo and V. Vassalos. The latter two are non-adjacent co-authors of R. Goldman's, but they have other relationships to R. Goldman in the database (e.g., they worked on the same research project). The lowest non-zero scores were given to both S. Venkatasubramanian and Y. Zhuge, representing the largest "separation" between R. Goldman and his co-authors. (On one paper, R. Goldman was first author and S. Venkatasubramanian was fourth, and on another, R. Goldman was second and Y. Zhuge was fifth.) In the original scheme, all of R. Goldman's co-authors tied for second place behind the element for R. Goldman himself.

6.7 Related Work

Database support for XML is a popular subject drawing much attention both in research and industry. In the research community, work on the XML-QL query language was the first effort to apply ideas from semistructured data research directly to XML [DFF+99a]. Like our work, XML-QL supports both an ordered an unordered model, though they do not model XML's graph structure as throughly as we do. Lorel's syntax is more similar to OQL's (or even SQL's) in comparison to XML-QL, though both languages are similar in expressive power.

Researchers at the University of Wisconsin are developing Niagara, a comprehensive XML query system. It includes mechanisms to identify relevant XML files across the Web and deal with remote data sources [NDea00]. Niagara also has investigated using a relational database management system to store XML [STH+99], in contrast to a native storage system like Lore.

In industry, XML activity is heating up as well. Most prominently, Microsoft led a coalition of companies to propose XQL, another XML query language [RLS98]. Currently, XQL is not as expressive as either XML-QL or Lorel, relying on a compact "URL-style" syntax for expressing queries. Companies such as Microsoft, Oracle, and IBM are working hard to enable their relational databases to publish and import XML, as well as allowing a database to be administered remotely via XML messages [SQL]. Finally, eXcelon Corporation, formerly Object Design, sells a data server for managing XML data, including support for native element-based storage and XQL queries [EXC].

Chapter 7

WSQ/DSQ: Combined Querying of Databases and the Web

Our work on semistructured data (Chapters 2 – 4) will be useful in the future as more and more data passes across the Internet as XML (Chapter 6). Proximity search (Chapter 5) enables powerful and intuitive keyword-based searches that exploit structure in traditional database systems. However, as of today we see two extremes of how data is managed and queried in practice. At one extreme, most enterprises store their operational data in relational database systems, and queries are issued via SQL. At the other extreme, search engines such as AltaVista and Google continually crawl and index millions of Web documents, but such systems only support simple keyword-based search.

In this chapter we propose a new approach that combines the existing strengths of traditional databases and Web searches into a single query system. *WSQ/DSQ* (pronounced "wisk-disk") stands for *Web-Supported (Database) Queries/Database-Supported (Web) Queries*. WSQ/DSQ is not a new query language. Rather, it is a practical way to exploit existing search engines to augment SQL queries over a relational database (WSQ), and for using a database to enhance and explain Web searches (DSQ). In terms of the matrix in Chapter 1, WSQ/DSQ is a bridge that couples two entries at opposite corners: it ties together Entry 1, representing expressive queries over structured data as supported by traditional database systems, with Entry 6, representing keyword-based search over unstructured data as supported by search engines. In terms of deployability, we believe WSQ/DSQ could be deployed more quickly than the other contributions in this thesis because it builds on mature technologies that are already used widely.

The basic architecture of WSQ/DSQ is shown in Figure 7.1. Each WSQ/DSQ instance queries one or more traditional databases via SQL, and keyword-based Web searches are routed to existing

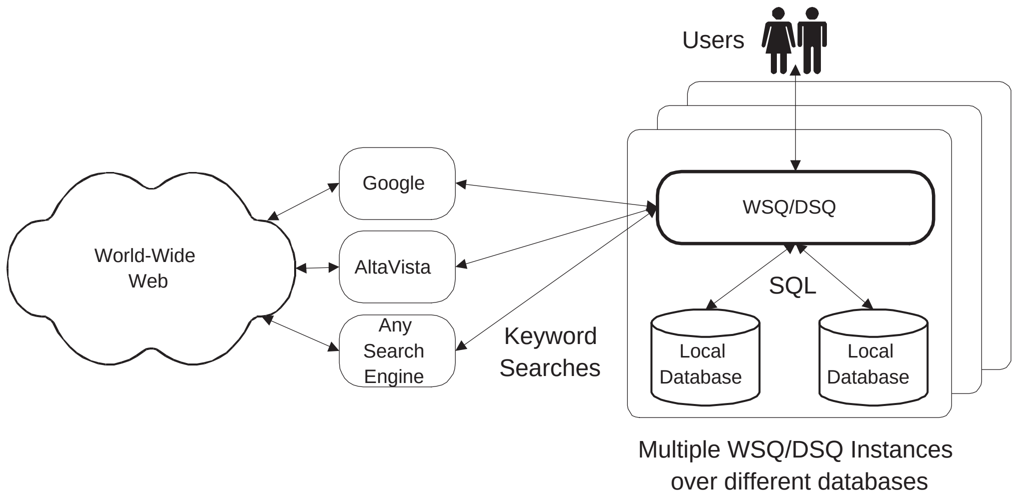


Figure 7.1: Basic WSQ/DSQ architecture

search engines. Users interacting with WSQ/DSQ can pose queries that seamlessly combine Web searches with traditional database queries.

As an example of WSQ (Web-Supported Database Queries), suppose our local database has information about all of the U.S. states, including each state's population and capital. WSQ can enhance SQL queries over this database using Web search engines to pose the following interesting WSQ queries (fully specifed in Section 7.2.1):

- Rank all states by how often they are mentioned by name on the Web.
- Rank states by how often they appear, normalized by state population.
- Rank states by how often they appear on the Web near the phrase “ four corners” .
- Which state capitals appear on the Web more often than the state itself?
- Get the top two URLs for each state.
- If Google (www.google.com) and AltaVista (www.altavista.com) both agree that a URL is among the top 5 URLs for a state, return the state and the URL.

WSQ does not perform any “ magic” interpretation, cleaning, or fltering of data on the Web. WSQ enables users to write intuitive SQL queries that automatically execute Web searches relevant to the query and combine the search results with the structured data in the database. With WSQ, we can easily write interesting queries that would otherwise require a signifcant amount of programming or manual searching.

DSQ (Database-Supported Web Queries) takes the converse approach, enhancing Web keyword searches with information in the database. For example, suppose our database contains information about movies, in addition to information about U.S. states. When a DSQ user searches for the keyword phrase "scuba diving", DSQ uses the Web to correlate that phrase with terms in the known database. For example, DSQ could identify the states and the movies that appear on the Web most often near the phrase "scuba diving", and might even find state/movie/scuba-diving triples (e.g., an underwater thriller filmed in Florida). DSQ can be supported using the system and techniques we present in this chapter, but we focus primarily on Web-supported queries (WSQ), leaving detailed exploration of DSQ for future work.

Much of the work presented in this chapter originally appeared in [GW00].

7.1 WSQ Overview

WSQ is based on introducing two *virtual tables*, `WebPages` and `WebCount`, to any relational database. A virtual table is a program that "looks" like a table to a query processor, but returns dynamically-generated tuples rather than tuples stored in the database. We will formalize our virtual tables in Section 7.2, but for now it suffices to think of `WebPages` as an infinite table that contains, for each possible Web search expression, all of the URLs returned by a search engine for that expression. `WebCount` can be thought of as an aggregate view over `WebPages`: for each possible Web search expression, it contains the total number of URLs returned by a search engine for that expression. We use `WebPages_AV` and `WebCount_AV` to denote the virtual tables corresponding to the AltaVista search engine, and we can have similar virtual tables for Google or any other search engine. By referencing these virtual tables in a SQL query, and assuring that the virtual columns defining the search expression are always bound during processing, we can answer the example queries above, and many more, with SQL alone.

While the details of WSQ query execution will be given later, it should be clear that many calls to a search engine may be required by one query, and it is not obvious how to execute such queries efficiently given typical search engine latency. One possibility is to modify search engines to accept specialized calls from WSQ database systems, but in our work we instead show how small modifications to a conventional database query processor can exploit properties of existing search engines.

When query processing involves many search engine requests, the key observations are:

- The latency for a single request is very high.
- Unless it explicitly supports parallelism, the query processor is idle during the request.
- Search engines (and the Web in general) can handle many concurrent requests.

Thus, for maximum efficiency, a query processor must be able to issue many Web requests concurrently while processing a single query. As we will discuss in Section 7.3, traditional (non-parallel) query processors are not designed to handle this requirement. We might be able to configure or modify a parallel query processor to help us achieve this concurrency. However, parallel query processors tend to be high-overhead systems designed for multiprocessor computers, geared towards large data sets and/or complex queries. In contrast, the basic problem of issuing many concurrent Web requests within a query has a more limited scope that does not require traditional parallelism for a satisfactory solution. To support our WSQ framework, we introduce a query execution technique called *asynchronous iteration* that provides low-overhead concurrency for external virtual table accesses and can be integrated easily into conventional relational database systems.

The main contributions discussed in this chapter are:

- A formalization of the WebPages and WebCount virtual tables and their integration into SQL, with several examples illustrating the powerful WSQ queries enabled by this approach, and a discussion of support for such virtual tables in existing systems (Section 7.2).

- *Asynchronous iteration*, a technique that enables non-parallel relational query processors to execute multiple concurrent Web searches within a single query (Section 7.3). Although we discuss asynchronous iteration in the context of WSQ, it is a general query processing technique applicable to other scenarios as well, and it opens up interesting new query optimization issues.

- Experimental results from our WSQ prototype (Section 7.4), showing that asynchronous iteration can speed up WSQ queries by a factor of 10 or more.

7.2 Virtual Tables in WSQ

For the purpose of integrating Web searches with SQL, we can can abstract a Web search engine through a virtual WebPages table:

WebPages(SearchExp, T1, T2, ..., Tn, URL, Rank, Date)

where SearchExp is a parameterized string representing a Web search expression. SearchExp uses "%1", "%2", and so on to refer to the values that are bound during query processing to attributes T1, T2, ..., Tn, in the style of printf or scanf [KR88]. For example, if SearchExp is "%1 near %2", T1 is bound to "Colorado" and T2 is bound to "Denver", then the corresponding Web search is "Colorado near Denver". For a given SearchExp and given bindings for T1, T2, ... Tn, WebPages contains 0 or more (virtual) tuples, where attributes URL, Rank, and Date are the values returned by the search engine for the search expression. The first URL returned by the search engine has Rank = 1, the second has Rank = 2, and so on. It is only practical to use WebPages in a query where SearchExp, T1, T2, ..., Tn are all bound, either by default (discussed below), through equality with a constant in the Where clause, or through an equi-join. In other words, these attributes can be thought of as "inputs" to the search engine. Furthermore, because retrieving all URLs for a given search expression could be extremely expensive (requiring many additional network requests beyond the initial search), it is prudent to restrict Rank to be less than some constant (e.g., Rank $<$ 20), and this constant also can be thought of as an input to the search engine.

A simple but very useful view over WebPages is:

WebCount(SearchExp, T1, T2, ..., Tn, Count)

where Count is the total number of pages returned for the search expression. Many Web search engines can return a total number of pages immediately, without delivering the actual URLs. As we will see, WebCount is all we need for many interesting queries.

Note that for both tables, not only are tuples generated dynamically during query processing, but the number of columns is also a function of the given query. That is, a query might bind only column T1 for a simple keyword search, or it might bind T1, T2, ..., T5 for a more complicated search. Thus, we really have an infinite family of infinitely large virtual tables. For convenience in queries, SearchExp in both tables has a default value of "%1 near %2 near %3 near ... near %n"[1]. For WebPages, if no restriction on Rank is included in the query, currently we assume a default selection predicate Rank $<$ 20 to prevent "runaway" queries.

Note also that virtual table WebCount could be viewed instead as a scalar function, with input parameters SearchExp, T1, T2, ..., Tn, and output value Count. However, since WebPages and other virtual tables can be more general than scalar functions—they can "return" any number of columns and any number of rows—our focus in this chapter is on supporting the general case.

[1] For search engines such as Google that do not explicitly support the "near" operator, we use "%1 %2 ... %n" as the default.

7.2.1 Examples

In this section we use WebPages and WebCount to write SQL queries for the examples presented informally in this chapter's introduction. In addition to the two virtual tables, our database contains one regular stored table:

```
States(Name, Population, Capital)
```

For each query, we restate it in English, write it in SQL, and show a small fraction of the actual result. The population values used for Query 2 are 1998 estimates from the U.S. Census Bureau [Uni98]. Queries 1– 5 were issued to AltaVista, and Query 6 integrates results from both AltaVista and Google. All searches were performed in October 1999.

Query 1: Rank all states by how often they appear by name on the Web.

```
Select Name, Count
From States, WebCount
Where Name = T1
Order By Count Desc
```

Note that we are relying on the default value of " %1" for WebCount.SearchExp. The first five results are:

```
<California, 4995016> <Washington, 4167056> <New York, 3764513>
<Texas, 2724285> <Michigan, 1621754> ...
```

Readers might be unaware that Texas and Michigan are the 2nd and 8th most populous U.S. states, respectively. Washington ranks highly because it is both a state and the U.S. capital; a revised query could exploit search engine features to avoid some false hits of this nature, but remember that our current goal is not one of " cleansing" or otherwise improving accuracy of Web searches.

Query 2: Rank states by how often they appear, normalized by state population.

```
Select Name, CountPopulation As C
From States, WebCount
Where Name = T1
Order By C Desc
```

Now, the first five results are:

<Alaska, 1149> <Washington, 733> <Delaware, 690> <Hawaii, 635> <Wyoming, 603> ...

Query 3: Rank states by how often they appear on the Web near the phrase "four corners".

```
Select Name, Count
From States, WebCount
Where Name = T1 and T2 = 'four corners'
Order By Count Desc
```

Recall that "%1 near %2" is the default value for WebCount.SearchExp when T1 and T2 are bound. There is only one location in the United States where a person can be in four states at once: the "four corners" refers to the point bordering Colorado, New Mexico, Arizona, and Utah. Note the dramatic dropoff in Count between the first four results and the fifth:

<Colorado, 1745> <New Mexico, 1249> <Arizona, 1095> <Utah, 994> <California, 215> ...

Query 4: Which state capitals appear on the Web more often than the state itself?

```
Select Capital, C.Count, Name, S.Count
From States, WebCount C, WebCount S
Where Capital = C.T1 and Name = S.T1 and C.Count > S.Count
```

In the following (complete) results, we again see some limitations of text searches on the Web—more than half of the results are due to capitals that are very common in other contexts, such as "Columbia" and "Lincoln":

<Atlanta, 1053868, Georgia, 958280> <Lincoln, 669059, Nebraska, 385991>
<Boston, 1409828, Massachusetts, 1006946> <Jackson, 1120655, Mississippi, 662145>
<Pierre, 663310, South Dakota, 283821> <Columbia, 1668270, South Carolina, 540618>

Query 5: Get the top two URLs for each state. We omit query results since they are not particularly compelling.

```
Select Name, URL, Rank
From States, WebPages
Where Name = T1 and Rank <= 2
Order By Name, Rank
```

Query 6: If Google and AltaVista both agree that a URL is among the top 5 URLs for a state, return the state and the URL.

```
Select Name, AV.URL
From States, WebPages_AV AV, WebPages_Google G
Where Name = AV.T1 and Name = G.T1 and AV.Rank <= 5 and G.Rank <= 5 and
      AV.URL = G.URL
```

Surprisingly, Google and AltaVista only agreed on the relevance of 4 URLs:

```
<Indiana, www.indiana.edu/copyright.html> <Louisiana, www.usl.edu>
<Minnesota, www.lib.umn.edu> <Wyoming, www.state.wy.us/state/welcome.html>
```

7.2.2 Support for virtual tables in existing systems

Both the IBM DB2 and Informix relational database systems currently support virtual tables in some form. We give a quick overview of the support options in each of these products, summarizing how we can modify our abstract virtual table definitions to work on such systems. (At the time of writing we understand that Oracle also expects to support virtual tables in a future release.) See [RP98] for more information about support for virtual tables in database products.

In DB2, virtual tables are supported through *table functions*, which can be written in Java or C [IBM]. A table function must export the number and names of its columns. Hence, DB2 cannot support a variable number of columns, so we would need to introduce a family of table functions WebPages1, WebPages2, etc. to handle the different possible number of arguments, up to some predetermined maximum; similarly for WebCount. To the query processor, a table function is an *iterator* supporting methods *Open*, *GetNext*, and *Close* [Gra93]. Currently, DB2 provides no "hooks" into the query processor for pushing selection predicates into a table function. At first glance, this omission apparently prevents us from implementing WebPages or WebCount, since both tables logically contain an infinite number of tuples and require selection conditions to become finite. However, DB2 table functions support parameters that can be correlated to the columns of other tables in a From clause. For example, consider:

```
Select R.c1, S.c3
From R, Table(S(R.c2))
```

In this query, S is a table function that takes a single parameter. DB2 will create a new table function iterator for each tuple in R, passing the value of c2 in that tuple to the *Open* method of S. (DB2 requires that references to S come after R in the From clause.) With this feature, we can implement WebPages and WebCount by requiring that SearchExp and the n search terms are supplied as table function parameters, either as constants or using the From clause join syntax shown in the example query above. In the case of WebPages, we must pass the restriction on Rank as a parameter to the table function as well.

Informix supports virtual tables through its *virtual table interface* [SBH98]. Unlike DB2, Informix provides hooks for a large number of functions that the DBMS uses to create, query, and modify tables. For example, in Informix a virtual table scan can access the associated Where conditions, and therefore can process selection conditions. However, the Informix query processor gives no guarantees about join ordering, even when virtual tables are involved, so we cannot be sure that the columns used to generate the search expression are bound by the time the query processor tries to scan WebPages or WebCount. Thus, Informix currently cannot be used to implement WebPages or WebCount (although, as mentioned earlier, WebCount could be implemented as a user-defined scalar function, which is supported in Informix).

7.3 WSQ Query Processing

Even with an ideal virtual table interface, traditional execution of queries involving WebCount or WebPages would be extremely slow due to many high-latency calls to one or more Web search engines. [CDY95] proposes optimizations that can reduce the number of external calls, and caching techniques [HN96] are important for avoiding repeated external calls. But these approaches can only go so far— even after extensive optimization, a query involving WebCount or WebPages must issue some number of search engine calls.

A *query plan* is a strategy for executing a query, typically a tree of operators that *scan*, *join*, *filter*, *aggregate*, and *sort* their inputs. As mentioned briefly in Section 7.2.2, query plans are usually executed using *iterators* that recursively drive query execution by asking operators to supply tuples. There is a cost associated with each operation in a query plan (usually based on predicted CPU and I/O time), and it is the job of the *query optimizer* to estimate the cost of different potential plans and choose the plan that it predicts will be the least expensive to execute [Gra93, GMUW00].

In many situations, the high latency of the search engine will dominate the entire execution time of the WSQ query. Any traditional non-parallel query plan involving WebCount or WebPages will be forced to issue Web searches sequentially, each of which could take one or more seconds, and the query processor is idle during each request. Since Web search engines are built to support many concurrent requests, a traditional query processor is making poor use of available resources.

Thus, we want to find a way to issue as many concurrent Web searches as possible during query processing. While a parallel query processor (such as Oracle, Informix, Gamma [DGS$^+$90], or Volcano [Gra90]) is a logical option to evaluate, it is also a heavyweight approach for our problem. For example, suppose a query requires 50 independent Web searches (for 50 U.S. states, say). To perform all 50 searches concurrently, a parallel query processor must not only dynamically partition the problem in the correct way, it must then launch 50 query threads or processes. Supporting concurrent Web searches during query processing is a problem of restricted scope that does not require a full parallel DBMS.

In the remainder of this section we describe *asynchronous iteration*, a new query processing technique that can be integrated easily into a traditional non-parallel query processor to achieve a high number of concurrent Web searches with low overhead. As we will discuss briefly in Section 7.3.2, asynchronous iteration is in fact a general query processing technique that can be used to handle a high number of concurrent calls to any external sources. As described in the following subsections, asynchronous iteration also opens up interesting new query optimization problems.

7.3.1 Asynchronous Iteration

Let us start with an example. Suppose in our relational database we have a simple table Sigs(Name), identifying the different ACM Special Interest Groups, called "Sigs" — e.g., SIGMOD, SIGOPS, etc. Now we want to use WebCount to rank the Sigs by how often they appear on the Web near the keyword "Knuth"[2]:

```
Select *
From Sigs, WebCount
Where Name = T1 and T2 = 'Knuth'
Order By Count Desc
```

Figure 7.2 shows a possible query plan for this query. For this plan, and for all other plans in this chapter, we assume an iterator-based execution model [Gra93] where each operator in the plan tree supports *Open*, *GetNext*, and *Close* operations. The *Dependent Join* operator requires each *GetNext* call to its right child to include a binding from its left child, thus limiting the physical join techniques that can be used to those of the nested-loop variety (although work in [HN96] describes

[2]Incidentally, the results (in order) from AltaVista are: SIGACT, SIGPLAN, SIGGRAPH, SIGMOD, SIGCOMM, SIGSAM. For all other Sigs, Count is 0.

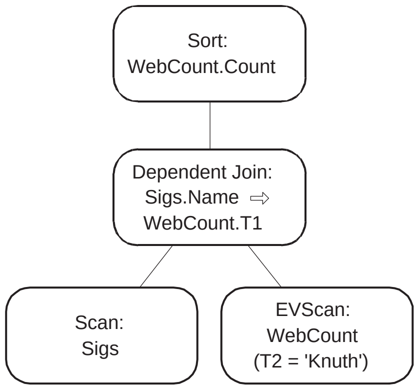


Figure 7.2: Query plan for Sigs $\bowtie$ WebCount

hashing and caching techniques that can improve performance of a dependent join). The *EVScan* operator is an external virtual table scan. We assume that we are working with a query processor that can produce plans of this sort— with dependent joins and scans of virtual tables— such as IBM DB2 (recall Section 7.2.2).

Without parallelism, EVScan performs a sequence of Web searches during execution of this query plan (one for each *GetNext* call), and the query processor may be idle for a second or more each time. Intuitively, we would like the query processor to issue many Web searches simultaneously, without the overhead of a parallel query processor. For this small data set— 37 tuples for the 37 ACM Sigs— we would like to issue all 37 requests at once. To achieve this behavior we propose asynchronous iteration, a technique involving three components:

1. A modified, asynchronous version of EVScan that we call *AEVScan*.
2. A new physical query operator called *ReqSync* (for " Request Synchronizer"), which waits for asynchronously launched calls to complete.
3. A global software module called *ReqPump* (for " Request Pump"), for managing all asynchronous external calls.

The general idea is that we modify a query plan to incorporate asynchronous iteration by replacing EVScans with AEVScans and inserting one or more ReqSync operators appropriately within the plan. AEVScan and ReqSync operators both communicate with the global ReqPump module. No other query plan operators need to be modified to support asynchronous iteration.

Now we walk through the actual behavior of asynchronous iteration using our example. Consider the query plan in Figure 7.3. In comparison to Figure 7.2, the EVScan has been replaced by an AEVScan, the ReqSync operator has been added, and the global ReqPump is used. When tuples

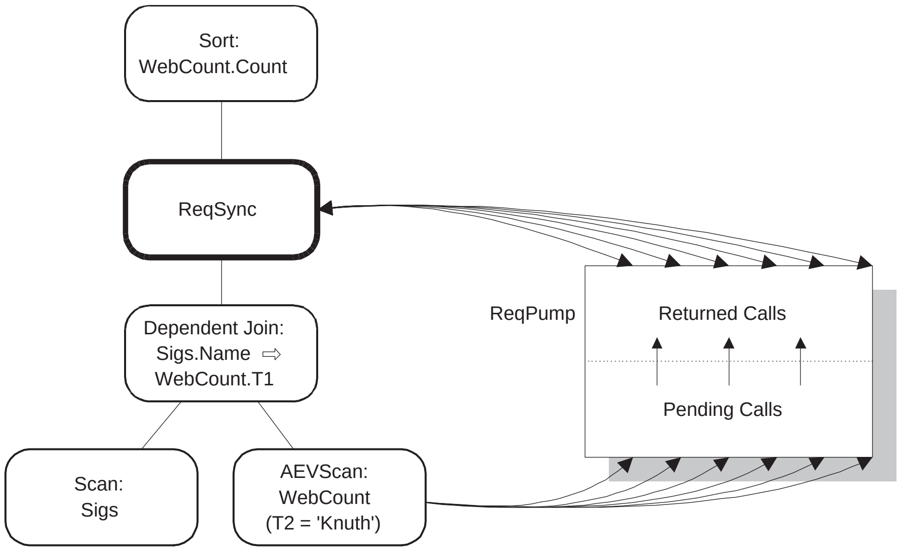


Figure 7.3: Asynchronous iteration

are constructed during query processing, we allow any attribute value to be marked with a special *placeholder* that serves two roles:

1. The placeholder indicates that the attribute value (and thus the tuple it's a part of) is incomplete.
2. The placeholder identifies a pending ReqPump call associated with the missing value— that is, the pending call that will supply the true attribute value when the call finishes.

Recall that all of our operators, including AEVScan and ReqSync, obey a standard iterator interface, including *Open*, *GetNext*, and *Close* methods. We now discuss in turn how the operators in our example query plan work.

The Scan and Sort operators are oblivious to asynchronous iteration. The Dependent Join (hereafter DJ) is a standard nested-loop operator that also knows nothing about asynchronous iteration. Now consider the AEVScan. When DJ gets a new tuple from Sigs, it calls *Open* on AEVScan and then calls *GetNext* with Sigs.Name. AEVScan in turn contacts ReqPump and registers an external call C with T1 = Sigs.Name and T2 = 'Knuth'. (C is a unique identifier for the call.) ReqPump is a module that issues asynchronous network requests and stores the responses to each request as they return. In the case of call C, the returned data is simply a value for Count; ReqPump stores this value in a hash table *ReqPumpHash*, keyed on C. To achieve concurrency, as soon as AEVScan registers its call with ReqPump, it returns to DJ (as the result of *GetNext*) one WebCount tuple

T where the Count attribute contains as a placeholder the call identifier C. DJ combines T with Sigs.Name and returns the new tuple to its parent (ReqSync).

Now let us consider the behavior of ReqSync. When its *Open* method is called from above by Sort, ReqSync calls *Open* on DJ below and then calls *GetNext* on DJ until exhaustion, buffering all returned (incomplete) tuples inside ReqSync. We choose this full-buffering implementation for the sake of simplicity, and we will revisit this decision momentarily. ReqSync needs to coordinate with ReqPump to fill in placeholders before returning tuples to its parent. The problem is a variation of the standard "producer/consumer" synchronization problem. Each ReqPump call is a producer: when a call C' completes (and its data is stored in ReqPumpHash), ReqPump signals to the consumer (ReqSync) that the data for C' is available. When signaled by ReqPump, ReqSync locates the incomplete tuple containing C' as a placeholder (using its own local hash table), and replaces C' with the Count value retrieved from ReqPumpHash. When ReqSync's *GetNext* method is called from above, if ReqSync has no completed tuples then it must wait for the next signal from ReqPump before it can return a tuple to its parent. Note that in the general case, tuples that do not depend on pending ReqPump calls may pass directly through a ReqSync operator.

In our simple implementation of ReqSync's *Open* method, all (incomplete) tuples generated by DJ are buffered inside ReqSync before ReqSync can return any (completed) tuples to its parent. In the case of very large joins it might make sense for ReqSync to make completed tuples available to its parent before exhausting execution of its child subplan. As with query execution in general, the question of materializing temporary results versus returning tuples as they become available is an optimization issue [GMUW00].

As we will show in Section 7.4, asynchronous iteration can improve WSQ query performance by a factor of 10 or more over a standard sequential query plan. However, there are still three important lingering issues that we will discuss in Sections 7.3.3, 7.3.4, and 7.3.5, respectively:

1. As seen in our example, an external call for WebCount always generates exactly one result tuple. But a call for WebPages may produce any number of tuples, including none, and the number of generated tuples is not known until the call is complete.

2. When a query plan involves more than one AEVScan, we must account for the possibility that an incomplete tuple buffered in ReqSync could contain placeholders for two or more different pending ReqPump calls.

3. We need to properly place ReqSync operators in relation to other query plan operators, both to guarantee correctness and maximize concurrency.

Monitoring and controlling resource usage is also an important issue when we use asynchronous iteration. So far we have assumed that during query execution we can safely issue an unbounded number of concurrent search requests. Realistically, we need to regulate the amount of concurrency to prevent a search engine from being inundated with an " unwelcome" number of simultaneous requests. Similarly, we may want to limit the total number of concurrent outgoing requests to prevent WSQ from exhausting its own local resources, such as network bandwidth. It is quite simple to modify ReqPump to handle such limits: we need only add one counter to monitor the total number of active requests, and one counter for each external destination. An administrator can configure each counter as desired. When a call is registered with ReqPump but cannot be executed because of resource limits, the call is placed on a queue. As resources free up, queued calls are executed.

7.3.2 Applicability of asynchronous iteration

Before delving into details of the three remaining technical issues outlined in the previous subsection, let us briefly consider the broader applicability of asynchronous iteration. Although this chapter describes asynchronous iteration in the specific context of WSQ, the technique is actually quite general and applies to most situations where queries depend on values provided by high-latency, external sources. More specifically, if an external source can handle many concurrent requests, or if a query issues independent calls to many different external sources, then asynchronous iteration is appropriate. Our WSQ examples primarily illustrate the first case (many concurrent requests to one or two search engines). As an example of the second case, asynchronous iteration could be used to implement a Web crawler: given a table of thousands of URLs, a query over that table could be used to fetch the HTML for each URL (for indexing and to find the next round of URLs). In this scenario, WSQ can exploit all available resources without burdening any external sources.

As mentioned earlier, if we try to use a parallel query processor to achieve the high level of concurrency offered by asynchronous iteration, then we would need to partition tables dynamically into many small fragments and spawn many query threads or processes. Issuing many threads can be expensive. For example, the highest performance Web servers do not use one thread per HTTP request; rather, many network requests are handled asynchronously by an event-driven loop within a single process [PDZ99]. By implementing the ReqPump module of asynchronous iteration in a similar manner, we can enable many simultaneous calls with low overhead. Nonetheless, as future work it would be interesting to conduct experiments comparing the performance of asynchronous iteration against a parallel DBMS for managing concurrent calls to external sources.

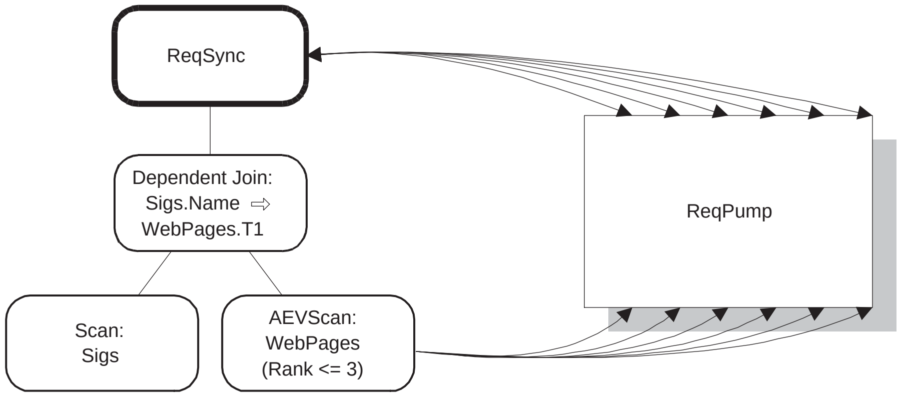

ReqSync
Dependent Join: Sigs.Name ⇨ WebPages.T1
ReqPump
Scan: Sigs
AEVScan: WebPages (Rank <= 3)

Figure 7.4: Query plan for Sigs ⋈ WebPages

7.3.3 ReqSync tuple generation or cancellation

The previous example (Figure 7.3) was centered on a dependent join with WebCount, which always yields exactly one matching tuple. But WebPages, and any other virtual table in general, may return any number of tuples for given bindings— including none. Because we want AEVScan to return from a *GetNext* call without waiting for the actual results, we always begin by assuming that exactly one tuple joins, then "patch" our results in ReqSync.

Consider the following query, which retrieves the top 3 URLs for each Sig.

```
Select *
From Sigs, WebPages
Where Name = T1 and Rank <= 3
```

For each Sig, joining with WebPages may generate 0, 1, 2, or 3 tuples. Assume a simple query plan as shown in Figure 7.4. As in our previous example, AEVScan will use ReqPump to generate 37 search engine calls, and ReqSync will initially buffer 37 tuples. Now consider what happens for a tuple T, waiting in a ReqSync buffer for a call C to complete. When C returns, there are three possibilities:

1. If C returns no rows, then ReqSync deletes T from its buffer.
2. If C returns 1 row, then ReqSync fills in the attribute values for T as generated by C.
3. If C returns n rows, where $n > 1$, then ReqSync dynamically creates $n - 1$ additional copies of T, and fills in the attribute values accordingly.

In our example, since all Sigs are mentioned on at least 3 Web pages, 111 tuples are ultimately produced by ReqSync.

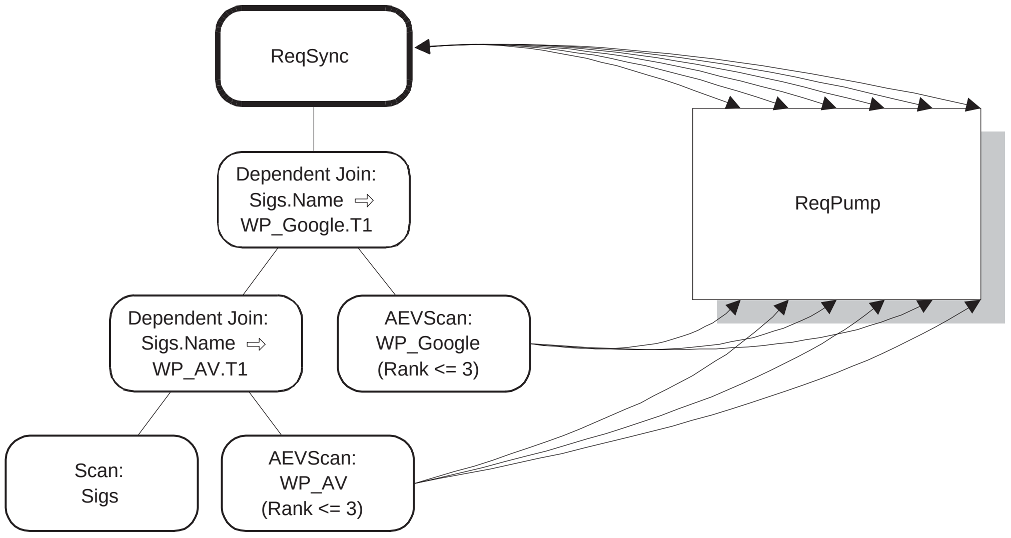

ReqSync
Dependent Join: Sigs.Name ⇨ WP_Google.T1
ReqPump
Dependent Join: Sigs.Name ⇨ WP_AV.T1
AEVScan: WP_Google (Rank <= 3)
Scan: Sigs
AEVScan: WP_AV (Rank <= 3)

Figure 7.5: Query plan for Sigs ⋈ WebPages_AV ⋈ WebPages_Google

7.3.4 Handling multiple AEVScans

Now let us consider query plans involving multiple AEVScans. For example, the following query finds the top 3 URLs for each Sig from two different search engines. [3]

```
Select *
From Sigs, WebPages_AV AV, WebPages_Google G,
Where Name = AV.T1 and Name = G.T1 and AV.Rank <= 3 and G.Rank <= 3
```

Figure 7.5 shows a query plan that maximizes concurrent requests. Note that there is only one ReqSync operator, not one for each AEVScan. The placement and merging of ReqSync operators is discussed in Section 7.3.5. In this plan, the bottom Dependent Join will generate 37 tuples, each with placeholders identifying a ReqPump call for WebPages_AV. The upper join will augment each of these tuples with additional placeholders corresponding to a ReqPump call for WebPages_Google. Hence, ReqSync will buffer 37 incomplete tuples, each one with placeholders for two different ReqPump calls.

The algorithm for tuple cancellation, completion, and generation at the end of Section 7.3.3 applies in this case as well, with a slight nuance: dynamically copied tuples (case 3 in the algorithm) may proliferate references to pending calls. For example, suppose one of the incomplete tuples T in the ReqSync buffer is waiting for the completion of two calls, indicated by two different

[3]The query actually finds all combinations of the top 3 URLs from each search engine, but it nonetheless serves to illustrate the point of this section.

placeholders: one for call C_A to AltaVista and the other for call C_G to Google. If C_A returns first, with 3 tuples, then ReqSync will make two additional copies of T. When copying T, references to pending call C_G are also copied. Once C_G returns, all tuples referencing C_G must be updated.

7.3.5 Query plan generation

Recall that converting a query plan to use asynchronous iteration has two parts: (1) EVScan operators are converted to AEVScans, and (2) ReqSync operators are added to the plan. In this section we describe an algorithm for placing ReqSync operators within plans. Our primary goal is to introduce a correct and relatively simple algorithm that: (1) attempts to maximize the number of concurrent Web searches; (2) attempts to maximize the amount of query processing work that can be performed while waiting for Web requests to be processed; and (3) is easy to integrate into existing query compilers. ReqSync operators can significantly alter the cost of a query plan, and the effects on query execution time will often depend on the specific database instance being queried, as well as the results returned by search engines. Fully addressing cost-based query optimization in the presence of asynchronous iteration is an important, interesting, and broad problem that is beyond the scope of our work.

We assume that the optimizer can generate plans with dependent joins [FLMS99] and EVScans, but knows nothing about asynchronous iteration; a plan produced by the optimizer is the input to our algorithm. We continue to assume an iterator model for all plan operators. We now describe the three steps in our placement of ReqSync operators: *Insertion*, *Percolation*, and *Consolidation*.

ReqSync Insertion

Recall that we first convert each EVScan operator in our input plan P to an asynchronous AEVScan. Next, a ReqSync operator is inserted directly above each AEVScan. More formally, for each AEVScan$_i$ in P, we insert ReqSync$_i$ into P as the parent of AEVScan$_i$. The previous parent of AEVScan$_i$ becomes the parent of ReqSync$_i$. This transformation is obviously correct since no operations occur between each asynchronous call and the blocking operator that waits for its completion.

ReqSync Percolation

Next, we try to move ReqSync operators up the query plan. Intuitively, each time we pull up a ReqSync operator we are increasing the amount of query processing work that can be done before blocking to wait for external calls to complete. Sometimes we can rewrite the query plan slightly to enable ReqSync pull-up. For example, if the parent of a ReqSync is a selection predicate that depends on attribute values filled in by ReqSync, we can pull ReqSync higher by pulling the selection predicate up first. Similarly, if a join depends on values filled in by ReqSync, we can rewrite the join as a selection over a cross-product and move the ReqSync above the cross-product.

Our actual algorithm is based on the notion of an operator O *clashing* with a ReqSync operator, in which case we cannot pull ReqSync above O. Let ReqSync$_i.A$ denote the set of attributes whose values are filled in by the ReqSync$_i$ operator as ReqPump calls complete, i.e., the attributes whose values are substituted with placeholders by AEVScan$_i$. We say that O *clashes* with ReqSync$_i$ iff:

1. O depends on the value of any attribute in ReqSync$_i.A$, or
2. O removes any attribute in ReqSync$_i.A$ via projection, or
3. O is an aggregation or existential operator

Case 1 is clear: an operator clashes if it needs the attributes filled in by ReqSync$_i$ to continue processing. Case 2 is a bit more subtle. If we project away placeholders before the corresponding calls are complete, then tuple cancellation or generation (Section 7.3.3) cannot take place properly, and extra tuples or incorrect numbers of duplicates may be returned. Case 3 is similar to case 2: aggregation (e.g., Count) and existential quantification require an accurate tally of incoming tuples.

For each ReqSync$_i$ in the plan, we repeatedly pull ReqSync$_i$ above any non-clashing operators. If an operator O does clash, we check to see if O is a projection or selection; if so, we can pull O above its parent first. Otherwise, if O is a clashing join, we rewrite it as a selection over a cross-product. Other similar rewrites are possible. For example, a set union operator must examine each complete tuple to perform duplicate elimination; we can rewrite this clashing operator as a "Select Distinct" over a non-clashing bag union operator. Our percolation algorithm clearly terminates since operators are only pulled up the plan. Also, the order in which we percolate ReqSync operators does not matter—the only potential effect is a different final ordering between adjacent ReqSync operators, something that is made irrelevant by ReqSync Consolidation, which we discuss next. We will illustrate the percolation algorithm through examples momentarily.

ReqSync Consolidation

After percolation, we may find that two or more ReqSync operators are now adjacent in the plan. At this point we can merge adjacent ReqSync operators since they perform the same overall function, and a single ReqSync operator can manage multiple placeholder values in tuples as discussed in

Section 7.3.4. When merging ReqSync$_i$ with ReqSync$_j$, ReqSync$_i.A \cup$ ReqSync$_j.A$ is the set of attributes that must be filled in by the new ReqSync operator.

Plan generation examples

We now show three examples demonstrating our ReqSync placement algorithm. We point out the performance gains asynchronous iteration can provide, along with some potential pitfalls of our current algorithm.

Example 7.1: Figure 7.6 shows how our ReqSync placement algorithm generates the query plan we saw earlier in Figure 7.5 for the Sigs $\bowtie$ WebPages_AV $\bowtie$ WebPages_Google query. We omit ReqPump from these (and all remaining) query plans. Figure 7.6(a) shows the input to the algorithm, a simple left-deep query plan without asynchronous iteration. Figure 7.6(b) shows the plan after ReqSync Insertion: the EVScans are converted to AEVScans and a ReqSync operator is inserted directly above each EVScan. Figure 7.6(c) shows the plan after ReqSync Percolation. We first move ReqSync$_1$ above both dependent joins, since neither join depends on any values returned by WebPages_AV (i.e., URL, Date, Rank). ReqSync$_2$ is then pulled above its parent dependent join. The final plan after ReqSync Consolidation is shown in Figure 7.6(d). With this plan, the query processor can process all 74 external calls (37 Sigs per join) concurrently.

This example demonstrates some interesting advantages of asynchronous iteration over possible alternatives. First, one might consider simply modifying the dependent join operator to work in parallel: change the dependent join to launch many threads, each one for joining one left-hand input tuple with the right-hand EVScan. While this approach will provide maximal concurrency for many simple queries, it prevents concurrency among requests from multiple dependent joins: the query processor will block until the first join completes. Another approach, as discussed in Section 7.3.2, is to use a (modified) parallel query processor for this query. However, performing both dependent joins in parallel requires a nontrivial rewrite to transform our 2-join plan into a 3-join plan where both dependent joins are children of a final "merging" join. □

Example 7.2: Consider the following query, where a cross-product with a meaningless table R is introduced for illustrative purposes:

```
Select *
From Sigs, WebCount_AV AV, R, WebCount_Google G
Where Name = AV.T1 and Name = G.T1
```

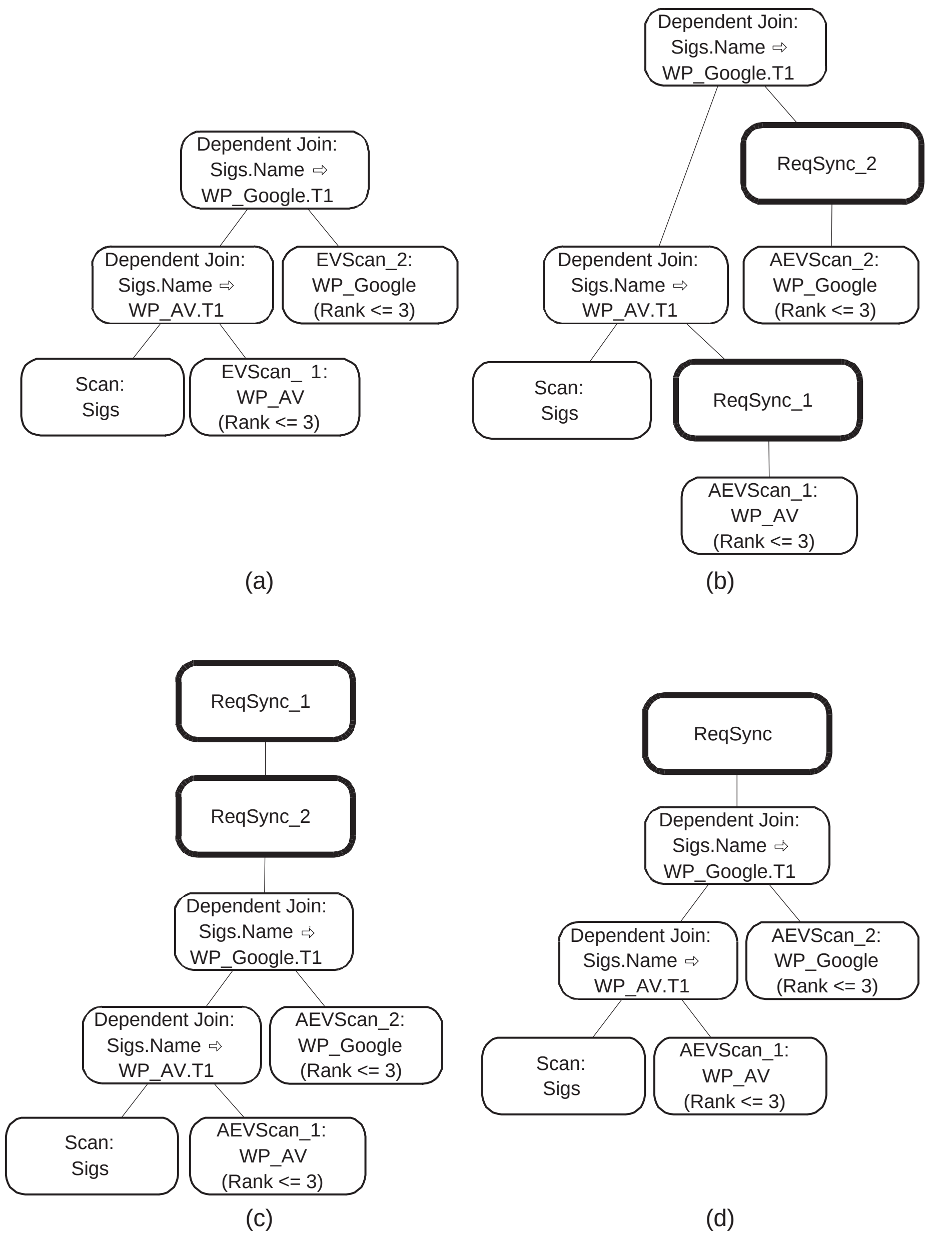

Dependent Join: Sigs.Name ⇨ WP_Google.T1
Dependent Join: Sigs.Name ⇨ WP_AV.T1
EVScan_2: WP_Google (Rank <= 3)
Scan: Sigs
EVScan_ 1: WP_AV (Rank <= 3)
(a)
Dependent Join: Sigs.Name ⇨ WP_Google.T1
ReqSync_2
Dependent Join: Sigs.Name ⇨ WP_AV.T1
AEVScan_2: WP_Google (Rank <= 3)
Scan: Sigs
ReqSync_1
AEVScan_1: WP_AV (Rank <= 3)
(b)
ReqSync_1
ReqSync_2
Dependent Join: Sigs.Name ⇨ WP_Google.T1
Dependent Join: Sigs.Name ⇨ WP_AV.T1
AEVScan_2: WP_Google (Rank <= 3)
Scan: Sigs
AEVScan_1: WP_AV (Rank <= 3)
(c)
ReqSync
Dependent Join: Sigs.Name ⇨ WP_Google.T1
Dependent Join: Sigs.Name ⇨ WP_AV.T1
AEVScan_2: WP_Google (Rank <= 3)
Scan: Sigs
AEVScan_1: WP_AV (Rank <= 3)
(d)

Figure 7.6: Generating the query plan for Sigs ⋈ WebPages AV ⋈ WebPages Google in Figure 7.5

Figure 7.7(a) shows the result of running our ReqSync placement algorithm over a left-deep input plan in which the cross-product with R is performed between the two virtual table dependent joins. With or without asynchronous iteration, this input plan is problematic: by performing the cross-product before the join with WebCount_Google, a straightforward dependent join implementation will send $|R|$ identical calls to Google for each Sig. Thus, incorporating a local cache of search engine results is very important for such a plan. Furthermore, when using asynchronous iteration with the plan in Figure 7.7(a), the cross-product with table R will generate $|R|$ copies of the incomplete tuples from WebCount_AV that must be buffered and then patched by ReqSync. Depending on the data, it may be preferable to use two ReqSync operators as shown in Figure 7.7(b). By doing so, we reduce the total number of attribute values to be patched by $|\mathsf{Sigs}| \cdot (|R| - 1)$, or roughly a factor of 2 for reasonably large $|R|$. On the down side, we will block after the first join, preventing us from concurrently issuing the Web requests for WebCount_Google. Had the cross-product with R been placed last in the original input plan, another alternative would be to place a single ReqSync operator above the dependent joins but below the cross-product.

This contrived example serves to illustrate the challenging query optimization problems that arise when we introduce AEVScan and ReqSync operators. Still, in many cases our simple ReqSync placement algorithm does perform well, as we will see in Section 7.4. □

Example 7.3: As a final example suppose that we also have a table CSFields(Name) containing computer science fields (e.g., "databases", "operating systems", "artificial intelligence", etc.). Consider the following query, which finds URLs that are among the top 5 URLs for both a Sig and a CSField.

```
Select S.URL
From Sigs, WebPages S, CSFields, WebPages C
Where Sigs.Name = S.T1 and CSFields.Name = C.T1 and S.Rank <= 5 and
      C.Rank <= 5 and S.URL = C.URL
```

An input query plan is shown in Figure 7.8(a). Note that the input plan is bushy, and the join at the the root of the plan may well be implemented as a sort-merge or hash join. After inserting the two ReqSync operators, we first pull them above the dependent joins. To pull the ReqSyncs above the upper join, we rewrite the join into a selection over a cross-product, as described in Section 7.3.5. (Because the join depends on attributes supplied by WebPages, we can't pull the ReqSync above it without the rewrite.) Figure 7.8(b) shows the final plan.

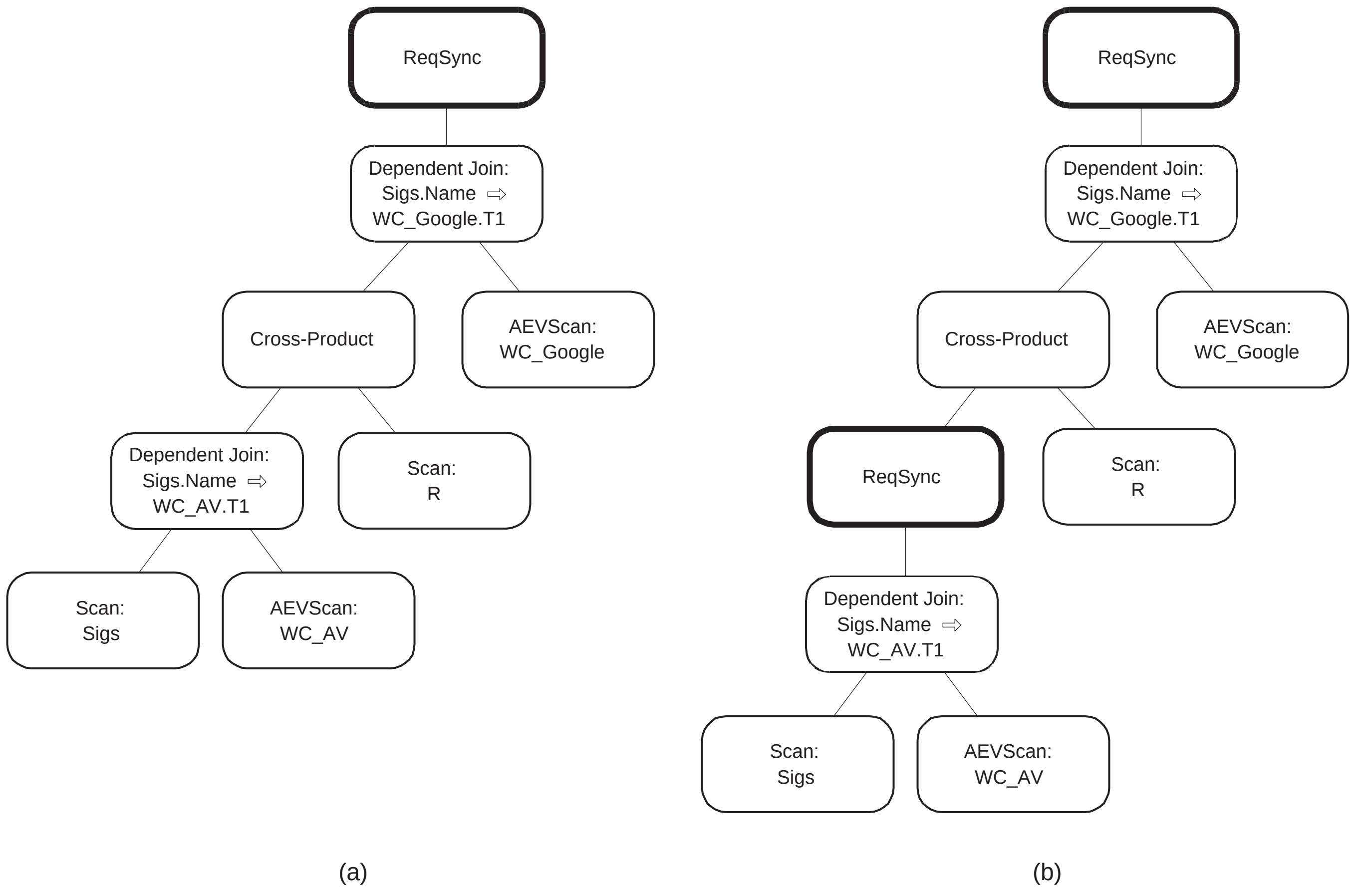

ReqSync
Dependent Join: Sigs.Name ⇨ WC_Google.T1
Cross-Product
AEVScan: WC_Google
Dependent Join: Sigs.Name ⇨ WC_AV.T1
Scan: R
Scan: Sigs
AEVScan: WC_AV
(a)
ReqSync
Dependent Join: Sigs.Name ⇨ WC_Google.T1
Cross-Product
AEVScan: WC_Google
ReqSync
Scan: R
Dependent Join: Sigs.Name ⇨ WC_AV.T1
Scan: Sigs
AEVScan: WC_AV
(b)

Figure 7.7: A query plan mixing two dependent joins with a cross-product

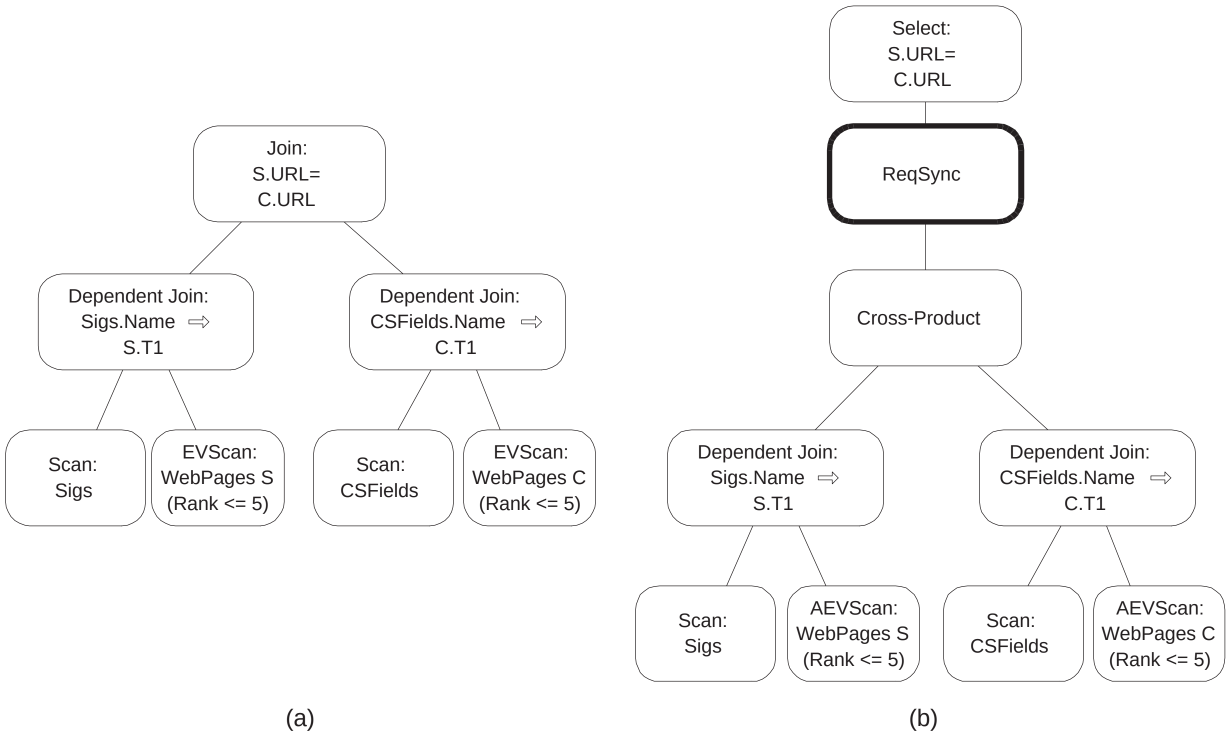

Join: S.URL= C.URL
Dependent Join: Sigs.Name ⇨ S.T1
Dependent Join: CSFields.Name ⇨ C.T1
Scan: Sigs
EVScan: WebPages S (Rank <= 5)
Scan: CSFields
EVScan: WebPages C (Rank <= 5)
(a)
Select: S.URL= C.URL
ReqSync
Cross-Product
Dependent Join: Sigs.Name ⇨ S.T1
Dependent Join: CSFields.Name ⇨ C.T1
Scan: Sigs
AEVScan: WebPages S (Rank <= 5)
Scan: CSFields
AEVScan: WebPages C (Rank <= 5)
(b)

Figure 7.8: Generating the query plan for query over Sigs and CSFields

In this query, given that the Sigs and CSFields tables are tiny, rewriting the join as a cross-product is a big performance win: it enables the query processor to execute all external calls (from both the left and right subplans) concurrently. However, in other situations, such as if the cross-product is huge, this specific rewrite could be a mistake.

This example illustrates one more important issue. Suppose that a Sig does not have any URLs on a given search engine. Indeed, assume for the moment that all Sigs have no URLs, so all Sig tuples generated will ultimately be canceled. In that case, pulling the ReqSync operator up as in Figure 7.8(b) results in an unnecessary cross-product between placeholder tuples for CSFields and WebPages, since ultimately the cross-product (and therefore the join) will be empty. In the general case, because AEVScan always returns exactly one matching tuple before the final result is known, a plan could perform unnecessary work— work that would not be done if the query processor waited for the true Web search result before continuing. □

To summarize, the above examples demonstrate how our ReqSync placement algorithm focuses on maximizing the number of concurrent external calls for any given query plan. If external calls dominate query execution time, then asynchronous iteration can provide dramatic performance improvements, as we demonstrate in Section 7.4. Nevertheless, there are several potential performance pitfalls that are best addressed by a complete cost-based query optimizer incorporating asynchronous iteration:

- Manipulating query plans to use asynchronous iteration may change their relative performance. Given two equivalent input plans A and B, where $Cost(A) < Cost(B)$, there is no guarantee that the asynchronous version of A will remain cheaper than the asynchronous version of B.
- The ReqSync operator buffers tuples, possibly proliferates them, and fills in missing attribute values. In some situations it is possible that the amount of work required by ReqSync offsets the advantages of asynchronous iteration.
- Asynchronous iteration assumes non-empty join results and continues processing, patching results later as necessary. If join results do turn out to be empty, then our "optimistic" approach will have performed more work than necessary.
- In order to pull ReqSync operators higher, we may move or rewrite operators in the input query plan, such as replacing joins with selections over cross-products. Additional work induced by these rewrites could offset the benefit of additional concurrency.

7.4 Implementation and Experiments

We have integrated the two WSQ virtual tables and our asynchronous iteration technique into a homegrown relational database management system called *RedBase*. (RedBase is constructed by students at Stanford in a course on DBMS implementation.) RedBase supports a subset of SQL for select-project-join queries, and it includes a page-level buffer and iterator-based query execution. However, it was not designed to be a high-performance system: the only available join technique is nested-loop join, and there is no query optimizer although users can specify a join ordering manually. Nevertheless, RedBase is stable and sophisticated enough to support the experiments in this section, which demonstrate the potential of asynchronous iteration. Our experiments show the considerable performance improvement of running WSQ queries with asynchronous iteration as opposed to conventional sequential iteration.

Measuring the performance of WSQ queries has some inherent difficulties. First, performance of a search engine such as AltaVista can fluctuate considerably depending on load and network delays beyond our control. Second, because of caching behavior at search engines beyond our control, repeated searches with identical keyword expressions may run far faster the second (and subsequent) times. To mitigate these issues, we waited at least two hours between queries that issue identical searches, which we verified empirically is long enough to eliminate caching behavior. Also, we performed our experiments late at night when the load on search engines is low and, more importantly, consistent.

In order to run many experiments without waiting hours between each one, we use *template* queries and instantiate multiple versions of them that are structurally similar but result in slightly different searches being issued. Consider the following template.

Template 1:

```
Select Name, Count
From States, WebCount
Where Name = T1 and WebCount.T2 = V1
```

V1 represents a constant that is chosen from a pool of different common constants, such as "computer", "beaches", "crime", "politics", "frogs", etc. For our experiments, we created 8 instances of the template by choosing 8 different constants from the pool. After timing all queries using asynchronous iteration, we waited two hours and then timed all queries using the standard query processor. For corroboration, we repeated the test with 8 new query instances.

The results for this template (and the two below) are shown in Table 7.1. For each template, we list the results of two runs. The times listed are the average execution time in seconds for the 8 queries, with and without asynchronous iteration. AltaVista is used for the first two templates; the third uses both AltaVista and Google. Experiments were conducted on a Sun Sparc Ultra-2 (2 x 200Mhz) 256MB RAM machine running SunOS 5.6. The computer is connected to the Internet via Stanford University's network.

Template 2:

```
Select Name, Count, URL, Rank
From States, WebCount, WebPages
Where Name = WebCount.T1 and WebCount.T2 = V1 and
    Name = WebPages.T1 and WebPages.T2 = V2 and WebPages.Rank <= 2
```

In this query template, we issue two searches for each tuple in States, one for WebCount and one for WebPages. When instantiating the template we wanted to ensure that V1 $\neq$ V2, so we

	Synchronous (secs)	Asynchronous (secs)	Improvement
Template 1			
Run 1 (8 queries)	23.13	3.88	6.0x
Run 2 (8 other queries)	32.8	3.5	9.4x
Template 2			
Run 1 (8 queries)	70.75	5.25	13.5x
Run 2 (8 other queries)	64.25	5.13	12.5x
Template 3			
Run 1 (8 queries)	122.5	6.25	19.6x
Run 2 (8 other queries)	76.13	4.63	16.4x

Table 7.1: Experimental results

selected 16 distinct constants to create 8 query instances. In our prototype system, the join order is always specified by the order of tables in the From clause, so for this query we joined States with WebCount, then joined the result with WebPages. Results are shown in Table 7.1.

Template 3: The following template is similar to the example in Section 7.3.4 (Figure 7.5), with the added constant V1. Again, we created 8 queries by instantiating V1 with constants, and results are shown in Table 7.1.

```
Select Name, AV.URL, G.URL
From Sigs, WebPages AV AV, WebPages Google G,
Where Name = AV.T1 and Name = G.T1 and AV.Rank <= 3 and G.Rank <= 3 and
    AV.T2 = V1 and G.T2 = V1
```

Our results show clearly that asynchronous iteration can improve the performance of WSQ queries by a factor of 10 or more. Of course, all of the example queries here are over very small local tables, so network costs dominate. These results in effect illustrate the best-case improvement offered by asynchronous iteration. For queries involving more complex local query processing over much larger relations, the speedup may be less dramatic, and the results of any such experiment would be highly dependent on the sophistication of the database query processor (independent of asynchronous iteration). Further, as illustrated in Section 7.3, complex queries may introduce optimization decisions that could have a significant impact on performance.

We have created a simple interface that allows users to pose limited queries over our WSQ implementation, available at http://www-db.stanford.edu/wsq.

7.5 Related Work

The techniques we know of that most closely relate to WSQ/DSQ are reported in [CDY95] and [DM97]. Written before the explosion of the World-Wide Web, [CDY95] focuses on execution and optimization techniques for SQL queries integrated with keyword-based external text sources. There are three main differences between [CDY95] and our work. First, they aim to minimize the number of external calls, rather than providing a mechanism to launch the calls concurrently. Nevertheless, some of techniques they propose are complementary to our framework and could be incorporated. Second, they assume that external text sources return search results as unordered sets, which enables optimizations that are not always possible when integrating SQL with (ranked) Web search results. Third, some of their optimizations are geared towards external text searches that return small (or empty) results, which we believe will be less common in WSQ given the breadth of the World-Wide Web. [DM97] discusses approaches for coupling a search engine with SQL, again without focusing on the World-Wide Web. A query rewrite scheme is proposed for automatically translating queries that call a search engine via a user-defined predicate into more efficient queries that integrate a search engine as a virtual table. While we also use a virtual table abstraction for search engines, [DM97] does not address the issue of high-latency external sources, which forms the core of much of this chapter.

The integration of external relations into a cost-based optimizer for LDL is discussed in [CGK89]. The related, more general problem of creating and optimizing query plans over external sources with limited access patterns and varying query processing capabilities has been considered in work on data integration, e.g., [HKWY97, LRO96, Mor88, RSU95, YLGMU99]. In contrast, we focus on a specific scenario of one type of external source (a Web search engine) with known query capabilities. [BT98] addresses the situation where an external source may be unavailable at a particular time: a query over multiple external sources is rewritten into a sequence of incremental queries over subsets of sources, such that the query results can be combined over time to form the final result. Although the asynchronous iteration technique we introduce shares the general spirit of computing portions of a query and filling in remaining values later, our technique operates at a much finer (tuple-level) granularity, it does not involve query rewriting, and the goal is to enable concurrent processing of external requests rather than handling unavailable sources.

Work in [UFA98] suggests a technique for improving response time of queries over high latency sources. In their approach, a query plan may be rescheduled or modified dynamically in response to delayed tuple arrival. In contrast, our asynchronous iteration technique optimistically assumes that external sources continually and immediately return tuples. These returned tuples, though potentially incomplete, allow the query processor to continue working without dynamic rescheduling. The statically placed ReqSync operators ensure that the final values returned by the external sources are handled correctly by the query processor.

As we saw in Section 7.3, we rely on *dependent joins* to supply bindings to our virtual tables when we integrate Web searches into a SQL query. Hence, previous work on optimizing and efficiently executing queries involving dependent joins is highly applicable. A general-purpose query optimization algorithm in the presence of dependent joins is provided in [FLMS99]. A caching technique that can be applied to improve the implementation of dependent joins is discussed in [HN96].

Much of the research discussed in this section is either preliminary or complementary to WSQ/DSQ. To the best of our knowledge, no previous work has taken our approach of enabling a non-parallel database engine to support many concurrent calls to external sources during the execution of a single query.

Chapter 8

Conclusions and Future Work

This thesis covers several contributions toward the challenging goal of unifying query functionality over structured, semistructured, and unstructured data.

We described our contributions to *Lore* (Chapter 2), enabling expressive queries over semistructured data— data that need not adhere to an explicitly declared, static schema. Our focus was on *DataGuides* (Chapter 3), which replace traditional schemas in a semistructured environment. A DataGuide is a dynamically generated and maintained summary of the structure of a semistructured database. DataGuides enable query formulation and optimization, and they also have been used both inside and outside of Stanford for other projects related to managing semistructured data.

Next, we focused on enabling interactive query and search sessions. In the context of a semistructured database, a user may wish to search, explore, and query a data set iteratively, until the desired data is reached. We described our model for enabling such interactive sessions in Chapter 4. As part of this model, basic single-keyword search for semistructured databases was introduced in Chapter 4. In Chapter 5, we described a more general approach to keyword-based search: we introduced *proximity search* in databases, building on the traditional information retrieval notion of "proximity." Our work in Chapter 5 applies to semistructured databases as well as to traditional structured databases.

In Chapter 6, we described how we extended our work on DataGuides and proximity search to apply to XML, the emerging standard for data interchange on the Web. XML is very similar to the original data models proposed for semistructured data, including the OEM model on which much of our work is based. But key differences needed to be addressed. Having the most impact, XML is inherently an ordered data model, whereas our algorithms for DataGuides and proximity search assumed unordered data.

Finally, in Chapter 7 we described *WSQ/DSQ*, a platform for efficient, tightly-coupled queries over existing relational database systems and Web search engines. Together, online relational databases and search engines combine to manage much of the Web's total data. WSQ/DSQ provides a platform for querying a relational database and searching the Web in a single framework, and the query processing techniques we introduced can improve overall performance by more than an order of magnitude over a naive approach.

In Section 8.1, we describe directions for future research related to each of the contributions of this thesis. Taking a longer-term view, much work remains to be done toward the ultimate goal of unifying all types of queries over all types of data. Thus, Sections 8.2 and 8.3 conclude this thesis with a discussion of two broad avenues of research that will move us closer to the vision of one ultimate data management system for all of the world's online data.

8.1 Future Work Related to Thesis Contributions

In the following five subsections, we describe potential future work corresponding to the topics of Chapters 3 – 7, respectively.

8.1.1 DataGuides (Chapter 3)

For many semistructured databases, the performance of DataGuide creation with respect to space and time is easily adequate. However, computing DataGuides for some databases is extremely expensive, especially for highly cyclic data. A challenging avenue of future work is to formalize the database characteristics that lead to good (or poor) DataGuide performance. Heuristics that quickly identify databases that may result in poor DataGuide performance would also be helpful.

While our work on Approximate DataGuides (ADGs) helps offset some of the DataGuide performance pitfalls, there are interesting potential extensions to the ADG work as well. In particular, among the several techniques we provided for approximation (object matching, suffix matching, and path-cycle matching), it may be possible to combine these techniques to generate the "best" approximation. An interesting avenue of research is to devise strategies that can efficiently analyze a database and select an approximation that will be quick to create and reasonably accurate as well, perhaps with statistical guarantees on the quality of approximation. Another interesting issue to pursue is ADG maintenance. While we can use a variation of our incremental DataGuide maintenance algorithm (Figure 3.6), there may be opportunities for better performance. For example, any ADG will remain an ADG after a database deletion (by definition). Another possibility is to use invalidation rather than incremental maintenance. An ADG may still be quite useful even if particular regions are marked “ invalid” due to updates. These regions could be recomputed in batch in the background, or perhaps the entire ADG could be regenerated when the percentage of invalid regions crosses some threshold.

8.1.2 Interactive Query and Search (Chapter 4)

At the highest level, we see much room for blurring the distinction between formulating and issuing a query. In our proposed model, we create a new DataGuide over the result of each query. However, these steps can be integrated. For example, simply adding a filtering condition to a DataGuide could trigger a query that automatically updates the DataGuide, perhaps eliminating DataGuide paths that are no longer relevant based on the given condition. To be effective in real-time, such tight integration requires high-performance coordination between query processing, DataGuide creation, and the user interface.

8.1.3 Proximity Search (Chapter 5)

In terms of performance, our hub-based indexing algorithm is open to further improvements. In particular, it may be possible to identify better heuristics for selecting hubs than those presented in [GSVGM98], especially when we can determine certain properties of the input graph. Abstractly, we introduced hubs because the space requirements of storing all K-neighborhoods on disk are enormous; if there were some way to effectively compress K-neighborhood storage on disk, query times could be improved dramatically. Another challenging direction is support for processing incremental changes to the underlying data; currently, the entire index must be recomputed.

As for functionality, we think it would be very interesting to add support for the Boolean operators *and*, *or*, and *not* to the context of searching databases. Reconsider the motivating example from Chapter 5, “ *Find* movie *Near* Travolta Cage.” Suppose a user really only wants movies near both *Travolta* and *Cage*. Currently, our proximity search treats all *Near* objects uniformly. Hence, if there were many “ Travolta” objects but only one “ Cage” object, a proximity query might highly rank a movie near all of the “ Travolta” objects, even if it is not near the “ Cage” object. Implementing a logical *and* requires either more sophisticated scoring functions or schemes for combining results from multiple proximity searches. At a higher level, there are significant opportunities for integrating proximity search with traditional database languages and models, as we discuss in Sections 8.2 and 8.3.

8.1.4 XML Support in Lore (Chapter 6)

Our work on XML has focused on the impact of order on DataGuides and proximity search. Another interesting avenue for future research is the impact of Document Type Definitions (DTDs).

We can build an Approximate DataGuide from a DTD (as described in Section 6.5), but it may also be interesting to combine DataGuides with DTDs: it is easy to envision a scenario where DTDs are available for specific portions of an XML database, but the overall database is still semistructured. We can build a DataGuide over the portions not governed by DTDs, with appropriate links to DTDs where appropriate. It is a challenge to coordinate these two structures to provide a unified view to users and applications, especially in the face of updates to the underlying XML data.

For proximity search, a DTD could be a useful tool for pruning the search space. For example, if we know that tags specified in a proximity search are restricted from ever appearing near each other, we can reduce the work required to perform the search.

At the highest level, a DTD may be sufficient to enable effective modeling and storage of XML in a traditional relational or object-oriented database system. As we discuss briefly in Section 8.3, an important long-term goal is to manage data effectively based on its inherent structure— not the particular encoding used to express it.

8.1.5 WSQ/DSQ (Chapter 7)

Asynchronous iteration was introduced as a technique to enable efficient WSQ queries, but it is a promising approach for many scenarios involving queries over external sources, not just Web search engines. For example, our WSQ approach could be used to compare prices of items stored in a local database at many different online vendors, obtaining concurrency across many different Web sites insetad of across many different calls to the same Web site.

Optimizing asynchronous query plans is a very challenging problem, and it requires a cost model that accounts not only for total work, but for response time as well. Further, modeling the latency and performance of external sources is inherently difficult. It would also be worthwhile to compare the performance of asynchronous iteration against a traditional parallel query processor. Alternatively, asynchronous iteration could be integrated into more radical approaches to query processing such as *eddies* [AH00], which continuously adapt and optimize query processing at runtime based on the delays of external sources.

Of course, given the title of our work, an obvious unexplored direction is to focus on DSQ, Database-Supported (Web) Queries. Keyword-based search still dominates searching on the Internet, and we feel there are many opportunities to enhance this experience by leveraging known relationships within a traditional database system. A software module built on top of WSQ could translate a user's keyword search into one or more WSQ queries and then rank the query results for presentation back to the user.

8.2 Language Integration

We have performed some initial steps toward unifying query functionality across relational databases, semistructured data, XML, and search engines, but today each type of system has its own language. SQL and Lorel are similar in spirit, but Lorel is most closely related to OQL[Cat94]— which is fast becoming obsolete. We must acknowledge that SQL is probably here to stay, and the best approach may be to integrate the most important features of Lorel into SQL. In the world of search engines, there is no standard query language; at best, there is the general acceptance that a keyword-based "query" is a boolean expression of keywords, using operators such as *and*, *or*, and *not*. (And as our work suggests, the *near* operator may well have a different meaning depending on whether the underlying data is a set of documents or an interconnected database.) An open question is whether we can integrate keyword-based search into SQL as well. Alternatively, perhaps SQL itself should be integrated or embedded within a larger, more comprehensive query language.

8.3 Model Integration

Relational data is set-oriented, based on relations containing sets of tuples. Semistructured data proposes an unordered, directed graph as its data model. XML data can be viewed as an ordered directed graph. To search engines, data is essentially a flat collection of documents. An ambitious goal is to provide a universal data model and storage management system that can support all such types of data yet not sacrifice any performance or expressive power provided by the native systems. The graph-based models of semistructured data may be universal enough to handle translation from other models, but the performance issue has not been addressed to significant depth. In other words, we cannot expect the world to translate their relational databases to XML (or OEM) if performance drops significantly. If we can export a universal data model yet still take advantage of specific structures and patterns of different data sets for higher performance, we may be able to slowly migrate the world toward a single view of data.

A strong benefit of the relational model is that the result of a query is itself a relation, which can be queried further. Work on Lore and other semistructured data management systems has enforced this idea as well, assuring that it is easy to query the result of an OEM or XML query. However, this approach becomes complicated in the world of keyword-based or other "fuzzy" searches: rankings and scores are an extremely important aspect of any such query result, and the system usually cannot make a simple binary decision of whether data is "in" or "out" of the result. There has been strikingly little work in unifying the worlds of set-based results with ranked and/or scored results (see [Fag96] for one important paper on this topic). However, we see this topic as being a critical component of merging traditional database queries with keyword-based search in an interactive, online setting.

Bibliography

[ABea98] Editors: V. Apparao, S. Byrne, and M. Champion et al. Document object model (DOM) level 1 specifcation, October 1998. W3C Recommendation available at http://www.w3.org/TR/REC-DOM-Level-1.

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

[ABS99] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufmann, San Francisco, California, 1999.

[AH00] R. Avnur and J. Hellerstein. Eddies: Continuously adaptive query processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 261– 272, Dallas, Texas, May 2000.

[AK97] N. Ashish and C.A. Knoblock. Wrapper generation for semi-structured internet sources. *SIGMOD Record*, 26(4):8– 15, 1997.

[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.

[BCSYDN$^+$99] L. Bouganim, T. Chan-Sine-Ying, T. Dang-Ngoc, J. Darroux, G. Gardarin, and F. Shea. MIROWeb: Integrating multiple data sources through semistructured data types. In *Proceedings of the Twenth-Fifth International Conference on Very Large Data Basees*, pages 750– 753, Edinburgh, Scotland, September 1999.

[BDFS97] P. Buneman, S. Davidson, M. Fernandez, and D. Suciu. Adding structure to unstructured data. In *Proceedings of the International Conference on Database Theory*, Delphi, Greece, January 1997.

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

[BDK92] F. Bancilhon, C. Delobel, and P. Kanellakis, editors. *Building an Object-Oriented Database System: The Story of O*$_2$. Morgan Kaufmann, San Francisco, California, 1992.

[BGMZ97] A. Broder, S. Glassman, M. Manasse, and G. Zweig. Syntactic clustering of the Web. In *Proceedings of the Sixth International World Wide Web Conference*, pages 391– 404, April 1997.

[BK89] E. Bertino and W. Kim. Indexing techniques for queries on nested objects. *IEEE Transactions on Knowledge and Data Engineering*, 1(2):196– 214, 1989.

[BT98] P. Bonnet and A. Tomasic. Partial answers for unavailable data sources. In *Proceedings of the Third International Conference on Flexible Query Answering Systems (FQAS)*, pages 43– 54, Roskilde, Denmark, May 1998.

[Bun97] P. Buneman. Semistructured data. In *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 117– 121, Tucson, Arizona, May 1997. Tutorial.

[Cat94] R.G.G. Cattell. *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 Twentieth International Conference on Very Large Data Bases*, pages 331– 341, Santiago, Chile, September 1994.

[CDSS98] S. Cluet, C. Delobel, J. Sim´eon, and K. Smaga. Your mediators need data conversion! In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 177– 188, Seattle, Washington, June 1998.

[CDY95] S. Chaudhuri, U. Dayal, and T. Yan. Join queries with external text sources: Execution and optimization techniques. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 410– 422, San Jose, California, 1995.

[CGK89] D. Chimenti, R. Gamboa, and R. Krishnamurthy. Towards an open architecture for LDL. In *Proceedings of the Fifteenth International Conference on Very Large Data Bases*, pages 195– 203, Amsterdam, The Netherlands, August 1989.

[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*, pages 203– 214, Bombay, India, August 1996.

[Com79] D. Comer. The ubiquitous B-tree. *Computing Surveys*, 11(2):121– 137, 1979.

[DFF$^+$99a] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A query language for XML. In *Proceedings of the Eighth International World Wide Web Conference (WWW8)*, Toronto, Canada, 1999.

[DFF$^+$99b] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. XML-QL: A query language for XML. In *Proceedings of the Eight International World-Wide Web Conference*, Toronto, Canada, May 1999.

[DGS$^+$90] D.J. DeWitt, S. Ghandeharizadeh, D.A. Schneider, A. Bricker, H.I. Hsiao, and R. Rasmussen. The Gamma database machine project. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):44– 62, 1990.

[Dij59] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269– 271, 1959.

[DM97] S. Deß loch and N. Mattos. Integrating SQL databases with content-specifc search engines. In *Proceedings of the Twenty-Third Internatial Conference on Very Large Databases*, pages 276– 285, Athens, Greece, August 1997.

[DR94] Shaul Dar and Raghu Ramakrishnan. A performance study of transitive closure algorithms. In *Proceedings of SIGMOD*, pages 454– 465, May 1994.

[EXC] eXcelon data server. http://www.odi.com/products/excelon/excelon dataserver.html.

[Fag96] R. Fagin. Combining fuzzy information from multiple systems. In *Proceedings of the Fifteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 216– 226, Montreal, Canada, June 1996. Tutorial.

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

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

[FLMS99] D. Florescu, A. Levy, I. Manolescu, and D. Suciu. Query optimization in the presence of limited access patterns. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 311– 322, Philadelphia, Pennsylvania, June 1999.

[Flo62] R. W. Floyd. Algorithm 97 (SHORTEST PATH). *Communications of the ACM*, 5(6):345, 1962.

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

[GMUW00] H. Garcia-Molina, J.D. Ullman, and J. Widom. *Database System Implementation*. Prentice Hall, Upper Saddle River, New Jersey, 2000.

[GMW99] R. Goldman, J. McHugh, and J. Widom. From semistructured data to XML: Migrating the Lore data model and query language. In *Proceedings of the 2nd International Workshop on the Web and Databases (WebDB '99)*, pages 25– 30, Philadelphia, Pennsylvania, June 1999.

[Goo61] I. J. Good. A causal calculus. *British Journal of the Philosophy of Science*, 11:305–318, 1961.

[Gra90] G. Graefe. Encapsulation of parallelism in the Volcano query processing system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 102– 111, Atlantic City, New Jersey, May 1990.

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

[GSVGM98] R. Goldman, N. Shivakumar, S. Venkatasubramanian, and H. Garcia-Molina. Proximity search in databases. In *Proceedings of the Twenty-Fourth International Conference on Very Large Data Bases*, pages 26– 37, New York, New York, August 1998.

[Gus97] D. Gusfeld. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, Cambridge, England, 1997.

[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.

[GW98] R. Goldman and J. Widom. Interactive query and search in semistructured databases. In *Proceedings of the International Workshop on the Web and Databases (WebDB '98)*, Valencia, Spain, March 1998.

[GW99] R. Goldman and J. Widom. Approximate DataGuides. In *Proceedings of the Workshop on Query Processing for Semistructured Data and Non-Standard Data Formats*, Jerusalem, Israel, January 1999.

[GW00] R. Goldman and J. Widom. WSQ/DSQ: A practical approach for combined querying of databases and the Web. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 285– 296, Dallas, Texas, May 2000.

[HGMC$^+$97] J. Hammer, H. Garcia-Molina, J. Cho, R. Aranha, and A. Crespo. Extracting semistructured information from the Web. In *Proceedings of the Workshop on Management of Semistructured Data*, pages 10– 17, Tucson, Arizona, May 1997.

[HKWY97] L. Haas, D. Kossmann, E. Wimmers, and J. Yang. Optimizing queries across diverse data sources. In *Proceedings of the Twenty-Third Internatial Conference on Very Large Databases*, pages 276– 285, Athens, Greece, August 1997.

[HN96] J.M. Hellerstein and J.F. Naughton. Query execution techniques for caching expensive methods. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 423– 434, Montreal, Canada, June 1996.

[Hop71] John Hopcroft. An n log n algorithm for minimizing the states in a finite automaton. In *The Theory of Machines and Computations*, pages 189– 196, New York, NY, 1971.

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

[IBM] IBM DB2 Universal Database SQL Reference Version 6. ftp://ftp.software.ibm.com/ps/products/db2/info/vr6/pdf/letter/db2s0e60.pdf.

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

[KR88] B. Kernighan and D. Ritchie. *The C Programming Language*. Prentice Hall, Upper Saddle River, New Jersey, 1988.

[LR88] T. Leighton and S. Rao. An approximate max-flow min-cut theorem for uniform multicommodity flow problems with applications to approximation algorithms. In *29th Annual Symposium on Foundations of Computer Science*, pages 422– 431, White Plains, New York, 1988.

[LRO96] A. Levy, A. Rajaraman, and J. Ordille. Querying heterogeneous information sources using source descriptions. In *Proceedings of the Twenty-Second International Conference on Very Large Databases*, pages 251– 262, Bombay, India, September 1996.

[LS00] H. Liefke and D. Suciu. XMILL: An efficient compressor for XML data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 153– 164, Dallas, Texas, May 2000.

[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.

[Mor88] K.A. Morris. An algorithm for ordering subgoals in NAIL! In *Proceedings of the Seventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 82– 88, Austin, Texas, 1988.

[MS99] T. Milo and D. Suciu. Index structures for path expressions. In *Proceedings of the International Conference on Database Theory*, pages 277– 295, January 1999.

[MW93] U. Manber and S. Wu. GLIMPSE: A tool to search through entire file systems. Technical report, Department of Computer Science, University of Arizona, October 1993. Available at URL ftp://ftp.cs.arizona.edu/glimpse/glimpse.ps.Z.

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

[MW99b] J. McHugh and J. Widom. Optimizing branching path expressions, June 1999. Available at ftp://db.stanford.edu/pub/papers/mp.ps.

[MW99c] J. McHugh and J. Widom. Query optimization for XML. In *Proceedings of the Twenty-Fifth International Conference on Very Large Data Bases*, pages 315– 326, Edinburgh, Scotland, September 1999.

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

[NAM98] S. Nestorov, S. Abiteboul, and R. Motwani. Extracting schema from semistructured data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Seattle, Washington, June 1998.

[NDea00] J. Naughton, D. DeWitt, and D. Maier et al. The niagra internet query system. Technical report, University of Wisconsin, 2000. Available at http://www.cs.wisc.edu/niagara/papers/NIAGRAVLDB00.v4.pdf.

[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, England, April 1997.

[Ora99] Oracle. Oracle interMedia text management. http://www.oracle.com/database/documents/intermedia text_mgmt ds.pdf, March 1999. White paper.

[PAGM96] Y. Papakonstantinou, S. Abiteboul, and H. Garcia-Molina. Object fusion in mediator systems. In *Proceedings of the Twenty-Second International Conference on Very Large Data Bases*, Bombay, India, 1996.

[PDZ99] V. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable web server. In *Proceedings of the USENIX 1999 Annual Technical Conference*, Monterey, CA, June 1999.

[PGGMU95] Y. Papakonstantinou, A. Gupta, H. Garcia-Molina, and J. Ullman. A query translation scheme for rapid implementation of wrappers. In *Proceedings of the Fourth International Conference on Deductive and Object-Oriented Databases*, pages 161– 186, Singapore, December 1995.

[PGMU96] Y. Papakonstantinou, H. Garcia-Molina, and J. Ullman. Medmaker: A mediation system based on declarative specifcations. In *Proceedings of the Internation Conference of Data Engineering, (ICDE '96)*, pages 132– 141, 1996.

[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.

[RLS98] J. Robie, J. Lapp, and D. Schach. XML query language (XQL). In *Proceedings of QL'98 – The Query Languages Workshop*, Boston, Massachusetts, December 1998. Papers available online at http://www.w3.org/TandS/QL/QL98/.

[RP98] B. Reinwald and H. Pirahesh. SQL open heterogenous data access. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 506– 507, Seattle, Washington, June 1998.

[RS97] M.T. Roth and P.M. Schwarz. Don't scrap it, wrap it! A wrapper architecture for legacy data sources. In *Proceedings of the Twenty-Third International Conference on Very Large Databases*, pages 266– 275, Athens, Greece, August 1997.

[RSU95] A. Rajaraman, Y. Sagiv, and J.D. Ullman. Answering queries using templates with binding patterns. In *Proceedings of the Fourteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 105– 112, San Jose, California, May 1995.

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

[SBH98] M. Stonebraker, P. Brown, and M. Herbach. Interoperability, distributed applications and distributed databases: The virtual table interface. *Data Engineering Bulletin*, 21(3):25– 33, 1998.

[SQL] What's New in SQL Server 2000. http://http://www.microsoft.com/sql/productinfo/sql2kover.htm.

[STH$^+$99] J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D. DeWitt, and J. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *Proceedings of the Twenty-Fifth International Conference on Very Large Data Bases*, pages 302– 314, Edinburgh, Scotland, September 1999.

[UFA98] T. Urhan, M. Franklin, and L. Amsaleg. Cost-based query scrambling for initial delays. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 130– 141, Seattle, Washington, June 1998.

[Uni98] United States Bureau of the Census. State population estimates and demographic components of population change: July 1, 1997 to July 1, 1998. http://www.census.gov/population/estimates/state/st-98-1.txt, December 1998.

[UW97] J.D. Ullman and J. Widom. *A First Course in Database Systems*. Prentice Hall, Upper Saddle River, New Jersey, 1997.

[Wag73] R. Wagner. Indexing design considerations. *IBM Systems Journal*, 12(4):351– 367, 1973.

[XML98] Extensible markup language (XML) 1.0, February 1998. W3C Recommendation available at http://www.w3.org/TR/1998/REC-xml-19980210.

[YLGMU99] R. Yerneni, C. Li, H. Garcia-Molina, and J. Ullman. Optimizing large join queries in mediation systems. In *Proceedings of the International Conference on Database Theory (ICDT)*, pages 348– 364, Jerusalem, Israel, January 1999.

[Zlo77] M. Zloof. Query by example. *IBM Systems Journal*, 16(4):324– 343, 1977.