DATA MANAGEMENT AND QUERY PROCESSING FOR SEMISTRUCTURED DATA

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

Jason George McHugh
March 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.

Dallan Quass

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

Traditional database management systems require all data to adhere to an explicitly specified, rigid schema. However, a large amount of the information available today is *semistructured* – the data may be irregular or incomplete, and its structure may evolve rapidly and unpredictably. It is difficult and inefficient to manage semistructured data using traditional relational, object-oriented, or object-relational database systems, which are designed and tuned for well-structured data.

This thesis describes *Lore*, a new database management system we developed for storing and querying semistructured data. The overall architecture of the Lore system contains many of the traditional database system components, but the fundamentally different nature of schema-less, semistructured data has required new techniques inside each component. This thesis covers our work in the overall system architecture, its query language, access methods, cost-based query optimizer, and view manager. We also describe a mechanism we developed by which Lore can dynamically and invisibly fetch and cache data from *external sources* during query processing.

Acknowledgments

Foremost, I thank my fiancee Kathi for her love and support. She keeps me balanced and focused and stood by me.

I thank my mother, Pauline, for caring so deeply about me. I can always feel her love.

I thank my father, Philip, for his advice and love. His support has always been very important to me.

I thank my brother, Flip. He motivated me when I was growing up and taught me that a healthy body is as important as a healthy mind. Flip always gives me a different perspective on things.

I thank my sister, Colleen, and her husband Chris. Their path through life and the family that they have built has given me a good idea of where I want to be someday in life.

I thank my nephew, Alex, for making my visits to Seekonk fun.

I am grateful to my advisor Jennifer Widom for teaching me what research is all about and for teaching me how to organize and convey my ideas effectively.

I thank the progenitors of Lore: Dallan Quass, Anand Rajaraman, and Hugo Rivero, for their foresight and for giving me the opportunity to work on a rewarding and exciting project.

I thank all the Lore developers: Brian Babcock, Andre Bergholz, Roy Goldman, Vineet Gossain, Kevin Haas, Matt Jacobson, Svetlozar Nestorov, Dallan Quass, Anand Rajaraman, Hugo Rivero, Michael Rys, Raymond Wong, Beverly Yang, Takeshi Yokokawa, for all their time and effort on the Lore code base. Lore is a product of everyone's hard work.

I thank my co-authors: Serge Abiteboul, Roy Goldman, Hector Garcia-Molina, Joachim Hammer, Dallan Quass, Michael Rys, Vasilis Vassalos, Jennifer Widom, Janet Wiener, Yue Zhuge, for their ideas, encouragement, and for teaching me how to do good research. I have learned a great deal from each of them.

Finally, I thank the members of the Stanford Database group for making the years here enjoyable, especially Roy Goldman, Vineet Gossain, Vasilis Vassalos, Sudarshan Chawathe, Tom Schirmer, Arturo Crespo, and Ben Werther.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Information available today can be viewed on a spectrum. At one end of the spectrum is *unstructured data*, for example a paragraph of flat text. At the other end of the spectrum is *structured data*, for example tables in a relational database. In the middle of the spectrum is *semistructured data*. Semistructured data has some inherent structure, but the data may be irregular and its structure may change quickly. Furthermore, some data may be missing, similar concepts may be represented using different types, and the structure at a given time may not be known fully. Much of the information contained in World-Wide Web pages is semistructured: data is embedded in HTML, it is varied and irregular, and the overall structure changes often. Semistructured data also arises when information is integrated from multiple, heterogeneous data sources, since different sources may represent the same type of data using different schemas [Com91, LMR90a, SL90].

Traditional database management systems (DBMSs), such as relational or object-oriented systems, are designed for structured data. For example, relational databases contain *tables* made up of *attributes* having fixed types. Object-oriented database systems rely on a specified class hierarchy. In both kinds of systems (and in hybrid *object-relational* systems) a schema must be defined before any data can be managed by the system.

Semistructured data can be forced into a traditional DBMS, however there are several major drawbacks:

- Considerable effort must be spent to devise a single, uniform schema that captures all of the data in the semistructured source.

- The semistructured data must be transformed into the well-structured form specified by the chosen schema before loading. For many applications this transformation can be costly and time-consuming.

- Additional effort is required later when the semistructured data is augmented or modified. Schema migration in traditional DBMSs is a well-known headache, involving changes from reorganization of the physical layout of data on disk to altering user or application queries.

- Queries over the well-structured encoding of semistructured data might not be natural to write or efficient to execute.

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

This thesis considers native management of semistructured data. Our system is based on a new data model and query language, also developed at Stanford. The data model is flexible enough to accommodate any data, regardless of its structure and without any prior structural knowledge. The query language supports intuitive query results, and a means of specifying queries without full knowledge of structure. It supports automatic coercion when types of data differ, does not generate errors when portions of the data are missing, and is not sensitive to inconsistencies in the data.

Based on this new data model and query language for semistructured data, we have developed a new kind of DBMS. Our complete DBMS for semistructured data has all the standard components: an application program interface (API), a query execution engine with query optimizer, low-level data storage, indexes, concurrency control, recovery, and a view manager. While some of these components look similar to their counterparts in relational or object-oriented DBMSs, many of them had to be modified considerably to operate in a schema-less, semistructured environment. An additional feature of our DBMS, enabled by the fact that it does not require a fixed schema, is the ability to efficiently bring in data from external sources in a dynamic, on-demand fashion.

1.1 Research Issues

Semistructured data management poses a number of new research issues. We briefly introduce many of the issues here, and then highlight our specific contributions in Section 1.2.

- An appropriate data model for semistructured data is required before a DBMS can be built. In this thesis we adopt the *Object Exchange Model* (*OEM*) a graph-based model for semistructured data introduced originally in the *Tsimmis* project at Stanford [PGMW95]. The model is designed to handle data that may be incomplete, as well as data with structure and type heterogeneity.

- Due to the properties of semistructured data—incompleteness and irregularity of the data, and rapid evolution of structure—unmodified traditional query languages are inappropriate. A query language for semistructured data should support: automatic coercion to relieve the user from strict typing; declarative *path expressions* describing traversals through the data graph that are powerful enough to be used when the structure is not fully known; data restructuring techniques to transform and replicate semistructured data; and a declarative update language. In this thesis we adopt the query language *Lorel* [AQM+97], developed at Stanford, which encompasses all of the necessary features.

- All DBMSs store data persistently on disk with a goal of efficient storage and efficient access to the data. Because OEM data is an arbitrary graph, it is not a simple matter to *cluster* data to support all possible access patterns efficiently. Our work has not focused on sophisticated storage schemes and their performance tradeoffs, relying instead on a relatively straightforward but workable scheme.

- A DBMS is a complex software system that can be divided into a set of components, each performing some facet of data management. These components, and how they work together, have been well-studied and understood in the context of relational, object-oriented, and object-relational database systems. While the overall architecture of a DBMS for semistructured data may be fairly traditional, each component must change to support the semistructured nature of the data. In this thesis we investigate how many of the components must change, with a specific focus on the query processing portion of the system.

- Multi-user support specialized for semistructured data has not been studied to any level of detail. While traditional logging and locking mechanisms [GR92] do work, more complex hierarchical locking systems [Ull88] might be adapted and applied. This area remains open for research.

- *Views* increase the flexibility of a DBMS by adapting the data to user or application needs [Ull89]. Defining views over semistructured data can be more complex than in traditional DBMSs. A view over semistructured data is defined over a graph and results (conceptually or physically) in another graph. In this thesis we introduce a view specification language suitable for semistructured data, and we investigate an incremental maintenance algorithm for materialized views over semistructured data.

- In this thesis we show that one of the advantages of a DBMS for semistructured data is the ability to integrate information brought in from external, possibly heterogeneous, information sources. Furthermore, it is possible to mix local and external data during query processing in a manner that is invisible to the user.

- A semistructured database may have some known portions of the data that are well-structured. Efficiently storing and querying a combination of structured and semistructured data is being considered by the *Ozone* project at Stanford [ALW99], while this thesis addresses purely semistructured data.

1.2 Contributions

We revisit some of the issues introduced in Section 1.1 and summarize in more detail the specific research contributions found in this thesis.

1.2.1 System Architecture

This thesis describes the overall architecture of *Lore*, a complete DBMS designed specifically for semistructured data. While the architecture of the Lore system is fairly traditional, many of the components of the system were modified to support semistructured data. For example, consider preprocessing and semantic checking of a query. In a relational system the schema is consulted to ensure the existence of tables and attributes, and to perform type checking over all query constructs. In contrast, a DBMS for semistructured data does not have a fixed schema and can do little to check the semantic correctness of a query.

Further, a query language for semistructured data is untyped, so type checking (even for simple predicates) is not desirable.

In this thesis we introduce the general architecture of the Lore DBMS as well as describe the unique characteristics of some components of Lore. We discuss the process that Lore goes through from the time a textual query is posed by a user or application to final query result generation. This work appeared originally in [MAG+97].

1.2.2 Query Optimization

A significant portion of this thesis is devoted to Lore's query optimization techniques, which enable the efficient execution of queries over semistructured data. The contributions of this thesis in the query optimization area include Lore's overall cost-based optimization framework, *query rewriting* techniques, and *post-optimization* techniques.

Lore's optimization framework includes *logical query plans*, *physical query plans*, a *plan enumeration strategy*, *statistics*, a *cost model*, and *cost formulas*. Once the basic framework is introduced, we extend the optimizer by considering optimization techniques for specific query language constructs. Since path expressions are the building blocks of most query languages for semistructured data [BDHS96, FFLS97, DFF+99], this thesis also investigates general query optimization techniques for path expressions in a semistructured environment. Portions of the query optimization work appeared originally in [MW99a].

1.2.3 View Management

Defining views over semistructured data can be more complex than in traditional DBMSs since a view is defined over a (possibly irregular) graph, and another graph is produced. One useful function of views over semistructured data is to introduce some structure to the data, to aid in query formulation and processing. To support this function, a view specification language must be able to add new nodes and edges that do not appear in the original data, and connect them in arbitrary ways to other view objects. In this thesis we introduce extensions to the Lorel query language for rich view specifications.

While both virtual and materialized views can be defined for semistructured data, only materialized views are considered in this thesis. Materialized view maintenance in the semistructured context is a very difficult problem. We introduce an incremental maintenance algorithm for a class of materialized views defined by our language. We also provide a performance study analyzing when incremental maintenance should be performed and when recomputing the view from scratch is preferred.

Work related to the view specification language first appeared in [AGM+97]. The work on materialization and maintenance of views appeared originally in [AMR+98].

1.2.4 External Data Management

This thesis introduces Lore's *external data manager*, and describes how it fits into the Lore system. The external data manager allows for the dynamic integration of data stored at external sources in a manner that is invisible to the user. We introduce optimizations in the external data manager's algorithms that reduce the amount of data transfered and number of calls that occur between Lore and external sources. This work appeared originally in [MW97].

1.3 Related Work

Here we describe work that is related to the general areas addressed by this thesis. More detailed discussion of work related to specific technical contributions appears in the relevant chapters.

The Lore project, which forms the basis for this thesis, began as a "supporting" project for *Tsimmis* [PGMW95, PGGMU95, PAGM96, PGMU96], a project in heterogeneous data integration at Stanford. In the Tsimmis architecture, *mediators* fetch and merge heterogeneous data from multiple, distributed sources in response to user's queries. Lore was designed originally to be a *lightweight* object repository that would be used by Tsimmis mediators for temporary data storage and query processing. Originally, "lightweight" referred both to the simple object model used by Lore (OEM), and to the fact that Lore was a lightweight system supporting single-user, read-only access. Lore quickly broke away from Tsimmis to take on a life of its own as a vehicle for research in semistructured data management. Lore has also evolved into a more traditional "heavyweight" DBMS in the functionality that it supports.

To date there are three other significant research projects that have studied semistructured data. The first project is centered around the query language *UnQL* [BDHS96], proposed by researchers at the University of Pennsylvania. While a working prototype of UnQL was never attempted, [BDHS96] gave a strong formalism for their proposed data model and query language, and introduced some interesting query constructs such as *traverse*, which allows for the restructuring of trees to an arbitrary depth. The second system, *Strudel*, is an infrastructure for Web-site management proposed primarily by researchers at AT&T [FFLS97]. A general discussion of the Strudel system appears in [FFK+99], and query optimization for the *StruQL* query language is discussed in [FLS98]. Many of the same researchers who created Strudel later introduced the third semistructured data project, which centers around the query language and accompanying data model for *XML-QL* [DFF+99]. This project represents the first work done by semistructured data researchers entirely in the context of the *eXtensible Markup Language* (*XML*) [BPSM98]. XML has emerged as a new standard for data representation and exchange on the World-Wide Web and has some obvious similarities with semistructured data. Researchers in semistructured data are now moving to XML, including the Lore project at Stanford [GMW99]. At the time of this writing several good tutorials or research overviews of semistructured data management have been produced, including [Abi97, Bun97, Suc97].

A tremendous amount of research has been done in the area of query optimization for DBMSs. Most of this work uses the relational data model and cannot be applied directly to the semistructured environment. However, some classic query optimization papers certainly influenced the optimization work in this thesis, including [SAC+79, Gra93]. In general, relational query optimization comes in two basic flavors. The first type of optimizer is a *rule-based optimizer*, such as the rule system in Starburst [PHH92], or the work in the *Coko-Kola* system [CZ98, CZ96]. Rule-based optimizers operate by defining internal algebras (or identifying common query constructs using code), and specifying declarative rules for transforming queries into equivalent queries with (what is presumed) a lower cost. The second type of optimizer is a *cost-based optimizer*, which creates and costs a set of low-level query plans. Our query optimizer is of the second type. Discussions of specific related work for the variety of optimizations that we perform appear in the appropriate chapters of this thesis.

Object-oriented query optimization research has focused primarily on defining an object algebra or calculus, and on equivalences that apply in such formalisms, e.g., [CD92, CCM96]. Less work has been done in true cost-based optimization for object-oriented queries, with most of it focusing on the restricted problem of simple path expressions [GGT96, SMY90, ODE95]. For example, to the best of our knowledge no published paper describes a cost-based optimizer for the entire OQL language.

A large body of work exists on the topic of database views, and on incremental maintenance of materialized views [BLT86, GMS93, GM95, RCK$^+$95, GL95]. With a few exceptions this work has focused on the relational data model. The view maintenance work most closely related to the work presented in this thesis appears in [Suc96, ZG98], but the work in this thesis lifts several constraints introduced in [Suc96, ZG98]. See Chapter 7 for further discussion.

Finally, much earlier work, such as *Model 204* and the *Multos* project [BRG88], supported storing data that is similar to semistructured data. Both systems were more limited than the work described in this thesis in terms of the query language capabilities and the type of data that was managed.

1.4 Thesis Outline

The remainder of the thesis proceeds as follows. Chapter 2 contains background information, including the OEM data model, several example OEM databases used throughout the thesis, the Lorel query language, and the overall architecture of the Lore system. Chapter 3 presents the framework for Lore's cost-based query optimizer. Chapter 4 investigates two query rewrite optimizations that Lore performs. A general optimization technique called *subplan caching* is introduced in Chapter 5. We consider the optimization of path expressions in Chapter 6. View management for semistructured data is described in Chapter 7. Chapter 8 describes Lore's *external data manager*. Conclusions and directions for future work are presented in Chapter 9.

Chapter 2

The Lore System

In this chapter we introduce background material on Lore, the DBMS for semistructured data that forms the basis for our work. We first introduce Lore's data model and query language. The data model of a DBMS defines the way in which users think about and interact with the data. (Frequently the physical storage of data closely follows the general data model of the system, but this approach is not required.) The query language provides a declarative means for users and applications to fetch data of interest from the database. We also introduce in this chapter the overall architecture of the Lore system, with further details on some components that interact closely with work described later in the thesis.

Lore's data model was introduced originally in [PGMW95] and the query language was described in [AQM+97]. The architecture of the Lore system, along with an overview of each of Lore's components, appeared originally in [MAG+97].

2.1 Introduction

Recall from Chapter 1 that semistructured data is data that may have some structure, but the structure and the data are not as rigid, regular, or complete as in traditional relational or object-oriented DBMSs. In this chapter we introduce background material necessary in order to discuss our specific contributions to data management and query processing for semistructured data.

First we introduce Lore's data model, which defines the way users think about the data. We use the *Object Exchange Model* (*OEM*), originally introduced in the *Tsimmis* project at Stanford [PGMW95]. OEM is a self-describing data model where schematic information in the form of labels is intermixed with the data. The version of OEM used in this thesis is slightly modified from the original OEM specification. In this chapter we introduce several sample OEM databases to illustrate the wide range of structural and type differences in OEM data. These databases are used for examples and performance experiments throughout the thesis.

Next we introduce Lore's query language, Lorel. Lorel is designed specifically for easy querying of semistructured data, with extensive automatic type coercion, powerful path expressions, and a familiar *select-from-where* syntax. This chapter provides a basic tutorial on Lorel to enable understanding of queries and query processing issues in subsequent chapters.

We then introduce the Lore system itself, our main vehicle for research in managing semistructured data. Lore is a full-featured DBMS that is available to the public and has been used in a variety of settings including Mitre Corporation, Northwestern University, and the MIT AI Lab. In designing Lore we found that the overall system architecture closely resembles the traditional architecture of a relational or object-oriented DBMS. However, many of the components that make up the Lore system differ considerably from components in traditional DBMSs because of the nature of semistructured data. In this chapter we discuss the overall architecture of the system and the unique characteristics of some of the components.

The remainder of this chapter proceeds as follows. Section 2.2 describes the data model OEM. Sample OEM databases are introduced in Section 2.3. The query language Lorel is introduced in Section 2.4. The overall architecture of the Lore system, along with definitions and notation used throughout the thesis, appears in Section 2.5. Related work is discussed in Section 2.6.

2.2 The Object Exchange Model

As a foundation for this thesis we adopt the Object Exchange Model (OEM) [PGMW95], a data model particularly useful for representing semistructured data. Intuitively, data represented in OEM can be thought of as a graph, with objects as the vertices and labels on the edges. More formally, in the OEM data model all entities are *objects*. Each object has a unique *object identifier* (*oid*). Some objects are *atomic* and contain a value from one of the disjoint basic atomic types, e.g., `integer`, `real`, `string`, `gif`, `html`, `audio`, `java`,

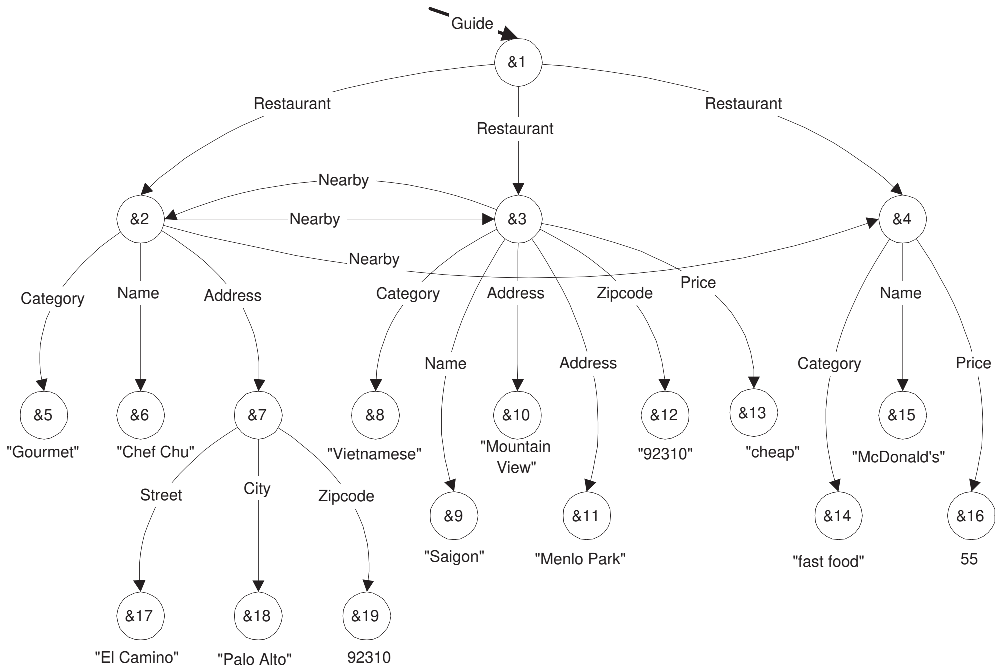

Figure 2.1: An example OEM database shown in graph form

etc. All other objects are *complex*; their value is a set of *object references*, denoted as a set of ⟨*label*, *oid*⟩ pairs. The labels are taken from the atomic type `string`.

In Figure 2.1, we show an example OEM database in graph form. This tiny database contains information about restaurants. The vertices in the graph are objects; the unique oid for an object is shown inside of each vertex, for example &5. Atomic objects have no outgoing edges and contain their atomic value. All other objects are complex objects and may have outgoing edges. For example, object &2 is complex and its *subobjects* are &5, &6, and &7. Object &17 is atomic and has value "El Camino". Note that object &1 has a special incoming edge labeled `Guide` which we describe in more detail next. We call the database represented by Figure 2.1 the *Guide* database.

OEM supports the concept of distinguished (object) *names*. There are many facets to the concept of a name in OEM:

- A name can be viewed as an alias for an object in the database. Objects with name aliases are referred to as *named objects*. For instance, *Guide* is the name of the object in Figure 2.1 that contains a collection of restaurants, i.e., object &1.

- A name serves as an *entry point* to the database. The only way objects can be accessed in queries is via paths originating from names.
- We require that all objects in the database are reachable from one of the names. (The rationale is that if an object becomes unreachable, no query will ever manage to access it, so the object might as well be garbage collected.) Hence, names also serve as *roots of persistence*: an object is persistent if it is reachable from one of the names.

Object &1 is the named object *Guide* in the database shown in Figure 2.1.

OEM can easily model relational, hierarchical, and graph-structured data. (Although the structure in Figure 2.1 is close to a tree, object &4 is "shared" by objects &1 and &2 and there is a directed cycle between objects &2 and &3.) OEM is well-suited to model semistructured data. Observe in Figure 2.1 that, for example: (i) restaurants have zero, one, or more addresses; (ii) an address is sometimes a string and sometimes a complex structure; (iii) a zipcode may be a string or an integer; and (iv) the zipcode occurs in the address for some restaurants and directly under restaurant for others.

Putting labels on edges rather than objects allows an object that is referenced from two different objects to have a different label from each referencing object. This feature is useful when an object participates in different relationships with different objects. For example, an object could be referenced as a "husband" of one object and a "father" of another. Note that labels on edges is a change made in Lore from the original OEM data model [PGMW95], where labels were attached to objects.

2.3 Sample OEM Databases

In this section we present a number of sample OEM databases that we will use throughout the thesis. We present them now to illustrate the wide variety of data that can be encoded in OEM. For each database we give only a small sample of the data or simply show the overall structure of the database. Of course all experiments run over these databases use much larger instances.

Guide database. Our first sample database, the *Guide* database containing restaurant information, was introduced in Figure 2.1 and discussed in Section 2.2.

Database Group database. The *DBGroup* database appears in Figure 2.2. The tiny illustrated database contains (fictitious) information about the Stanford Database Group,

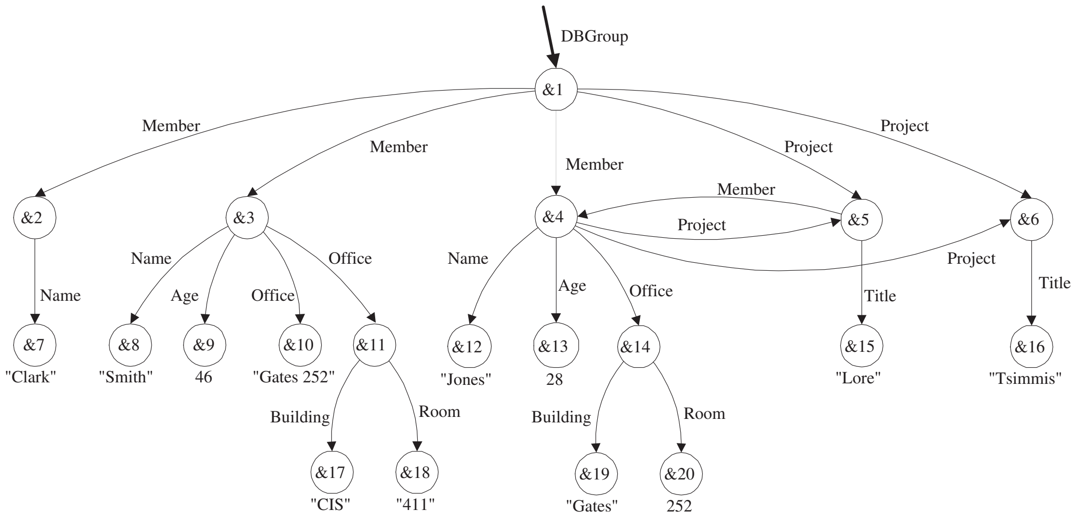

Figure 2.2: Small (fictitious) sample of the Database Group database

including group members and projects. Note that this data is graph-structured and contains cycles: A member of the group can point to a project that he works on and a project can point back to that member. The small database shown in Figure 2.2 is a representative sample of a much larger real database about the Stanford Database Group, which also contains information about publications and is very highly interconnected. The real database has 3,633 objects and 4,797 edges.

Movies database. The *Movies* database contains information about movies made in 1997. The database was created by combining information from many sources including the Internet Movie Database (`http://www.imdb.com`). The database contains facts about 1,970 movies, 10,260 actors and actresses, plot summaries, directors, editors, writers, etc., as well as multimedia data such as still photos and audio clips. The database is semistructured and very cyclic. Figure 2.3 contains a portion of the *structural summary* (see Section 2.5.3) for the Movies database. The full summary of the Movies database is quite large, however the subset shown in Figure 2.3 is sufficient for examples in this thesis.

Movie Store database. The *MovieStore* database is a synthetically generated database containing information that a corporate office for a big video rental chain might maintain about their stores and competitor's stores, or that might be maintained within the movie industry itself. The database contains fictitious information about movies, stores that rent

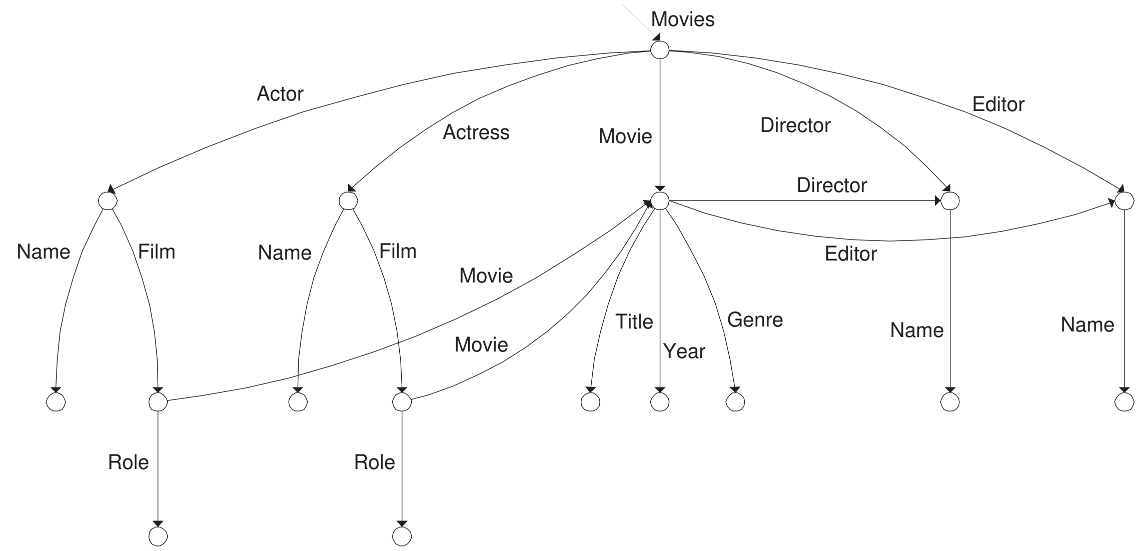

Figure 2.3: Structure of the Movies database

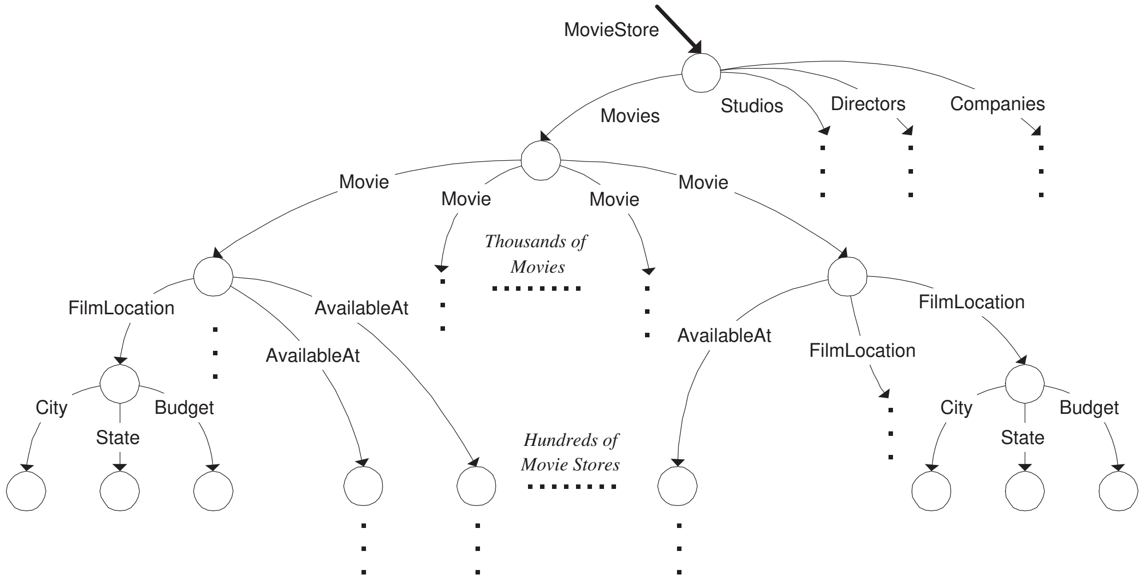

Figure 2.4: Structure and cardinality of the Movie Store database

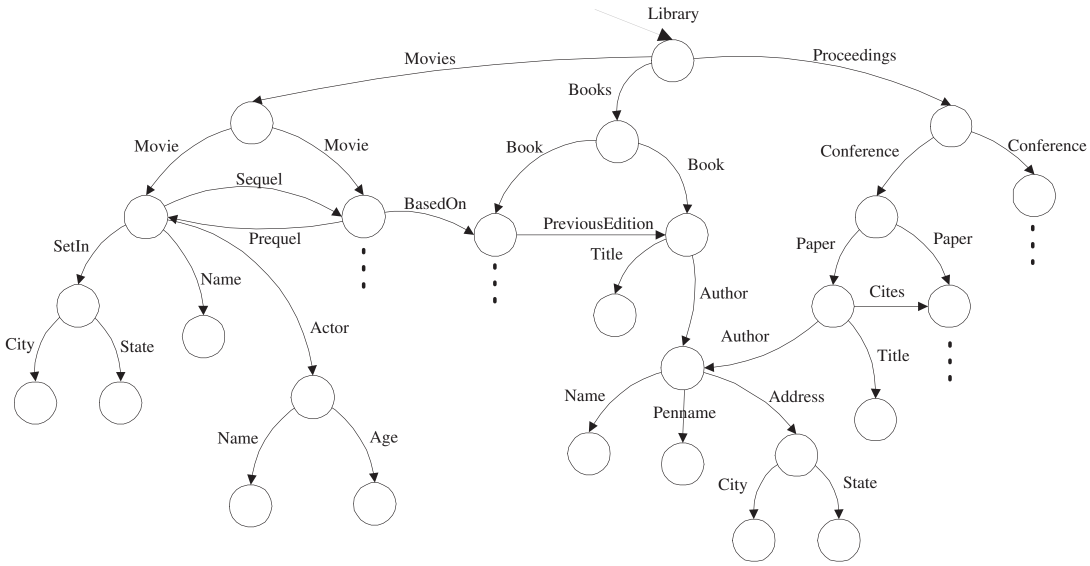

Figure 2.5: Structure of the Library database

and sell the movies, companies that own the stores, and people that work for the companies or have participated in making a movie. The general structure of a subset of the data is shown in Figure 2.4, along with some indications of database size. The shape of the data in Figure 2.4 is important and occurs often in real-world data. The graph alternates from being very wide (thousands of movies) to narrow (hundreds of video stores). This shape results in many possible paths leading to movie stores, but many fewer distinct movie stores.

Library database. The *Library* database is a synthetically generated database containing information that might be stored in a database maintained by a library. The data includes information about books or conference proceedings available for checkout, movies available for viewing, etc. The general structure of the Library database is shown in Figure 2.5. The actual shape and size of the database are dependent on parameters that are set when an instance of the database is generated.

Book database. The *BookDB* database is a small fictitious database containing information about books and authors. A representative sample of the Book database is shown in Figure 2.6. This database is used for clarity in examples where a larger and more complex database is not needed.

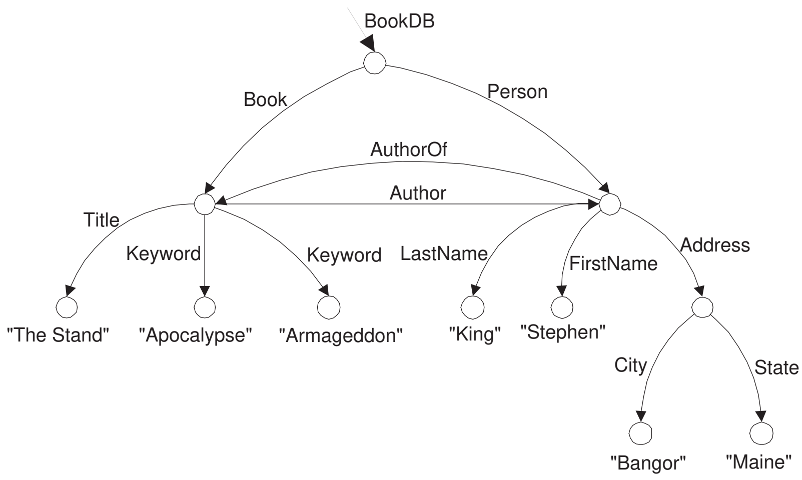

Figure 2.6: Small sample of the the Book database

2.4 Query Language

In this section we provide an overview of the query language Lorel. Historically, the first version of Lorel was introduced in [QRS+95a], but was later updated in [AQM+97] to the version used in this thesis. Our overview is given in a tutorial style, consisting of a sequence of example queries. Each query is intended to be executed over the Guide database shown in Figure 2.1.

2.4.1 Path Expressions

A *path expression* specifies a traversal through an OEM database. In its simplest form a path expression is a dot-separated sequence of labels that begins with a name (recall Section 2.2). When a path expression is applied to (or matched against) a database it results in all possible paths through the database that match the sequence of edge labels. A path expression by itself is a valid Lorel query.

Example 2.4.1 Consider the following query consisting of a single path expression. The query returns the set of objects matched at the end of the path expression.

```
Guide.Restaurant.Address
```

Although the query can be executed in many ways as we will discuss later, we can imagine that it first locates the named object `Guide`, corresponding to object &1 in Figure 2.1. From that object the set of all `Restaurant` subobjects {&2, &3, &4} are discovered. From each of these the set of all `Address` subobjects is fetched. The query result is the set of objects {&7, &10, &11}. □

Object variables can be introduced to break a single long path expression into multiple path expressions. Each variable is *bound* to the set of objects that result from its path expression. A path expression can then begin either with a name, or with a variable that has been defined by a previous path expression.

Example 2.4.2 The following path expression is equivalent to the path expression appearing in Example 2.4.1.

```
Guide.Restaurant r, r.Address a
```

Again the expression can be evaluated in many ways, but let us say that it starts from the named object `Guide`, then finds all `Restaurant` subobjects and places them into the variable r. From each object bound to r the set of all `Address` subobjects is placed into variable a. For this path expression r is bound to {&2, &3, &4} and a is bound to {&7, &10, &11}. Unlike Example 2.4.1, in Lorel this construct is not a query on its own. □

Introducing object variables can make a query easier to read, but more importantly it allows a path expression to *branch* and explore two separate subgraphs in a database.

Example 2.4.3 The following path expression uses variable r twice in order to discover both the name and categories of a restaurant.

```
Guide.Restaurant r, r.Name n, r.Category c
```

In the Guide database in Figure 2.1, when r is bound to &2 then n is bound to &6 and c is bound to &5. □

2.4.2 Select-From-Where Queries

Lorel supports the standard *select-from-where* syntax found in SQL and other query languages. In Lorel the *from* clause contains a list of path expressions. The *where* clause provides filtering of paths matched in the *from* clause by using selection conditions or subqueries. The *select* clause constructs the final result. The result of a *select-from-where* query is a set of objects.

Example 2.4.4 Our first *select-from-where* query introduces several features of Lorel that are useful when querying semistructured data. Lorel supports many syntactic shortcuts and in subsequent examples we show additional ways to express the query.

```
select r.Address
from Guide.Restaurant r
where r.Name = "Chef Chu"
```

This query finds the set of addresses for the restaurant with name "Chef Chu". The *from* clause binds to variable r the set of all restaurants. The *where* clause filters out restaurants not named "Chef Chu". The *select* clause is called once for each object bound to r that satisfies the *where* clause, and it extracts the set of `Address` subobjects for r. The result for this query is the set of set-of `Address` objects.

Lorel is designed to handle the case where a restaurant may have no `Name` subobjects, a single `Name` subobject, or a set of `Name` subobjects. The *where* clause "`where r.Name = "Chef Chu"`" is interpreted in Lorel to be "`where exists n in r.Name: n="Chef Chu"`", allowing the query to execute properly in all situations. Note that by default this means that a restaurant matches the *where* clause if there exists at least one `Name` subobject whose value matches the predicate. This implicit existential interpretation is appropriate in most situations, and can be expressed explicitly by the user. If universal quantification is the desired interpretation then "`where for all n in r.Name: n="Chef Chu"`" can be used instead. □

Lorel supports extensive automatic coercion since the types of atomic objects may differ. For example, the prices of entrees in a larger version of our Guide database may sometimes be string values and other times real or integer values, and a comparison between the two values `4.37` and `"4.37"` should return `true`. In Lorel there is a hierarchy of types that supports coercion from one type to a more specific type in the hierarchy as long as no information is lost. In general, values are coerced into comparable types before performing any comparisons. If two values cannot be coerced into comparable types then the predicate evaluates to `false`. More details on coercion appear in Section 2.5.2.

Example 2.4.5 The following query is equivalent to Example 2.4.4:

```
select Guide.Restaurant.Address
```

```
from Guide.Restaurant
where Guide.Restaurant.Name = "Chef Chu"
```

The general philosophy in Lorel is that path expressions that are within the same scope and have the same prefix are bound to the same objects in the database. The three path expressions in this example share the common prefix "`Guide.Restaurant`", so it is equivalent to assigning a variable to the path expression prefix, as in Example 2.4.4. □

Like SQL and other query languages, the *where* and *select* clauses in Lorel can contain complex expressions including subqueries.

Example 2.4.6 Multiple path expressions in the *select* clause in Lorel are very common. Consider the following query:

```
select r.Name, r.Address
from Guide.Restaurant r
```

This query returns the set of `Name` and `Address` subobjects for each restaurant. To ensure that the connection between a set of names and addresses for a single restaurant is not lost, a new object with the label `Restaurant` is created in the result for each binding of r. Subobjects of this new object are the set of names and addresses for the current r. The following two queries are equivalent to the one above:

```
select (select n from r.Name n), (select a from r.Address a)
from Guide.Restaurant r

select oem(Restaurant:(select n from r.Name n) union
                      (select a from r.Address a))
from Guide.Restaurant r
```

Recall that a *select-from-where* query results in a set of objects. In the first equivalent query, the construction of the restaurant objects in the result is implicit by the semantics of Lorel. The second equivalent query makes this construction explicit using the `oem` construct, which creates a new object with incoming edge `Restaurant` and with children that are a result of the `union` expression. If a restaurant doesn't have any `Address` (or `Name`) subobjects, then the corresponding subquery in both of the two queries above results in the empty set, and no `Address` (or `Name`) subobjects will exist in the result for that particular restaurant. □

Example 2.4.7 Our final *select-from-where* example shows the ease of specifying non-trivial queries in Lorel.

```
select Guide.Restaurant.Address, Guide.Restaurant.Name
where count(Guide.Restaurant.Entree) > 14
```

This query retrieves the set of addresses and names for each restaurant that serves more than fourteen entrees. Note that this query does not have a *from* clause. In Lorel the *from* clause can be omitted, and it will be generated automatically based on the *select* clause. In the simplest case, when the *select* clause contains a single path expression, the path expression in the *select* is copied into the *from* clause. When the *select* clause contains more than a single path expression, we extract the greatest common path expression prefix from all expressions in the *select* clause and place the prefix in the *from* clause. For the query in this example the *from* clause generated by the semantics of Lorel is "`from Guide.Restaurant`". This query also uses one of the five standard aggregation operations (`count`, `sum`, `avg`, `min`, `max`) which, like most query languages, Lorel supports. □

2.4.3 Path Patterns

Lorel supports more powerful path expressions than those described in Section 2.4.1, useful when the structure of the data is irregular or unknown. Lorel's regular expression operators in path expressions, along with "wildcards" discussed below, allow the user to specify "path patterns" instead of exact paths. The allowed regular expression operators are:

- "*" is used to match 0 or more edges in the graph. For example, "`r(.Nearby)*`" will match 0 or more `Nearby` edges in the graph beginning from the object bound to r.
- "+" is used to match 1 or more edges in the graph. It is similar to "*" except it must always match at least one edge.
- "?" indicates an optional edge. For example, "`r(.Address)? x`" will bind x to all `Address` subobjects of r as well as bind x to the object in r. This expression allows the `Address` edge to be skipped.
- "|" indicates a choice of edges. For example, "`r(.FaxNumber|.TelephoneNumber) n`" will bind n to both the `FaxNumber` and `TelephoneNumber` subobjects of r.

Note that in the examples above the regular expression operator is applied only to single labels. In general, regular expression operators can be applied to a sequence of labels. For example, "`x(.Friend.Mother)* y`" will match zero or more instances of the path "`Friend.Mother`". Regular expression operators in path expressions can be nested in any manner in Lorel, however the current Lore implementation does not support the full generality of Lorel's regular expression operators. All of the regular expression operators are implemented, but they can only be applied to single labels or to a list of labels separated by "|".

Lorel also supports *label completion*, allowing a label to be replaced partially or completely with the *label wildcard* "%", which matches 0 or more characters. For example, the label `Author%` would match `Author`, `AuthorName`, `Authors`, etc. The path expression "`Guide.Restaurant.%`" would match any subobject of a restaurant. The path expression "`(.%)*`", which can be written in shorthand as "`.#`", matches zero or more edges with any label, and is a common construct in queries.

2.4.4 Updates

Lorel also contains a declarative update language. Using the update language, it is possible to create and destroy names, create new atomic or complex objects, modify the values of existing atomic objects, and create, delete, and replace edges. There is no explicit object deletion since deletion occurs implicitly when an object becomes unreachable.

Example 2.4.8 The following update adds a restaurant's city as a direct subobject of the restaurant object whenever the city is Palo Alto or Menlo Park.

```
update r.City += c
from Guide.Restaurant r, r.Address.City c
where c = "Palo Alto" or c = "Menlo Park"
```

The *from* and *where* clauses are the same as in a normal *select-from-where* statement. The binding of variables r and c in the *from* clause and the evaluation of the *where* clause are done before performing any updates. This update adds a new edge (the `+=` operator) from objects bound to r to objects bound to c with the label `City`. Value updates and edge deletion have a similar flavor. □

2.4.5 Disjunctive and Conjunctive Normal Forms

We define *disjunctive normal form* (*DNF*) and *conjunctive normal form* (*CNF*) for Lorel *select-from-where* queries. First, we require that no "shortcuts" are used:

- The query must be fully specified with a *from* clause.
- Common subpaths must be eliminated from path expressions and replaced with a common variable (recall Section 2.4.2).
- All variables appearing in the *where* clause must be explicitly quantified by either existential or universal quantification.

We also require that all quantification in the *where* clause appears at the outermost scope. That is, all `and` and `or` operators in the *where* clause appear within the scope of all quantifiers.

In addition to these requirements, a *select-from-where* query is in DNF only when the *where* clause (if present) is expressed as a series of disjuncts, where each disjunct is either an atomic predicate or the conjunction of atomic predicates. A *select-from-where* query is in CNF only when the *where* clause is expressed as a series of conjuncts, where each conjunct is either an atomic predicate or the disjunction of atomic predicates.

A few algorithms in this thesis require either DNF or CNF form. Currently it is an open question whether all (or even most) Lorel queries can be converted to DNF or CNF form.

2.4.6 Summary and Status

We have introduced many key features of the Lorel language, including path expressions, the *select-from-where* syntax, aggregation operations, set operations, object construction, and the update language. All of these features are supported in the current Lore system. Additional implemented features of Lorel include path variables, arithmetic operations, and Skolem functions. Other features in the design of Lorel but not yet implemented include *group-by* and *order-by* clauses, external functions and predicates, the full generality of `not` in the *where* clause, subqueries in the *from* clause, and some simple query constructs such as `Abs` (for absolute value of an integer or real) and `Element` (for returning an arbitrary element in a set). More details on the query language can be found in [AQM$^+$97]. The full syntax for Lorel is provided in Appendix A.

2.4.7 Notation and Terminology

We conclude this section by introducing notation and terminology related to Lorel that will be used in the remainder of the thesis.

Subpath. A *subpath* is a portion of a path expression that will be treated as a unit. It consists of a single label, or a portion of a path expression with a regular expression operator applied to it. We do not further decompose within regular expression operators. For example, given path expression "`A(.B|(.C)*)+`" its two subpaths are "`A`" and "`(.B|(.C)*)+`".

Path expression components. Every path expression can be broken up into a list of *path expression components*. Each component is a triple $\langle$*source variable*, *subpath*, *destination variable*$\rangle$. The *source variable* is a variable defined in a previous path expression component or the special variable name "`Root`". Root corresponds to a known database object from which all named objects are reachable. The *subpath* is as defined above. The *destination variable* is bound to objects that are descendants via the subpath of objects bound to the source variable. We say the source variable in a path expression component is said to *feed* the subpath, and the subpath likewise feeds the destination variable. We write a path expression component as either $\langle r, \mathit{Name}, n\rangle$ or "`r.Name n`".

Path Expression. A *path expression* p is a list of path expression components: c_1, c_2, ..., c_n. Each component c_i has the following restrictions:

- The source variable for c_i must appear as the destination variable of a path expression component c_j where $1 \leq j < i$.
- The destination variable for c_i cannot appear as the destination variable for any other component in p.

Mapping a Lorel path expression to our formal definition is straightforward. Each subpath of the path expression becomes a new path expression component with newly created (if required) source and destination variables connecting adjacent subpaths in the original path expression. The one exception is when the subpath corresponds to a *name* (Section 2.2), in which case the source variable is the special symbol `Root` described above. For example, we break the path expression "`Guide.Restaurant.Name`" into three path expression components since it contains three subpaths. Three new variables are introduced: g between

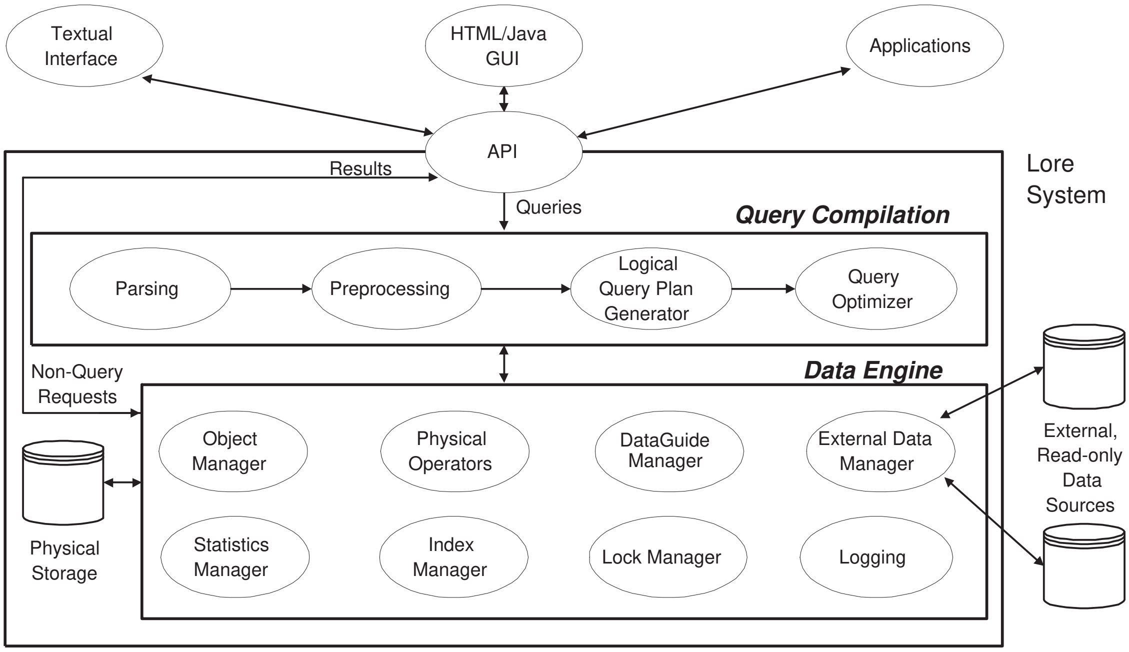

Figure 2.7: Lore architecture

`Guide` and `Restaurant`, r between `Restaurant` and `Name`, and n for the destination variable of the `Name`'s. The path expression components are $\langle$`Root`, `Guide`, $g\rangle$, $\langle g$, `Restaurant`, $r\rangle$, and $\langle r$, `Name`, $n\rangle$.

Branching path expression. A *branching path expression*, introduced informally in Section 2.4.1, is a path expression containing at least one variable that appears at the beginning of more than one path expression component.

General path expression. A *general path expression* is a path expression containing at least one subpath that has a regular expression operator applied to it.

2.5 System Architecture

Given as a basis the data model and query language presented in the previous sections, we now introduce the Lore system architecture and discuss the interaction between components of the system.

The basic architecture of the Lore system is depicted in Figure 2.7. Access to the Lore system is through a variety of applications or directly via the Lore application program interface (API). There is a simple textual interface, primarily used by the system developers, but suitable for learning system functionality and exploring small databases. The graphical interface, the primary interface for end users, provides powerful tools for browsing query results, a *DataGuide* feature for seeing the structure of the data and formulating simple queries "by example" (Section 2.5.3), a list of frequently-asked queries, and mechanisms for viewing the multimedia atomic types such as `video`, `audio`, and `java`. These two interface modules, along with other applications, communicate with Lore through the API.

The *query compilation* layer of the Lore system consists of the *parser*, *preprocessor*, *logical 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 "canonicalizes" a Lorel query by eliminating Lorel shortcuts and translating the query into a form expected by later components of the query compiler (see Section 2.5.1). A logical query plan is generated from the transformed parse tree and then passed to the query optimizer. A logical query plan consists of *logical operators* that describe the generic steps required to answer a query. The query optimizer consists of three subcomponents (not shown in Figure 2.7): the *query rewrite module*, *physical plan enumerator*, and *post-optimization module*. Details on each of these subcomponents appear in Chapter 3. Overall, the query optimizer component produces a *physical query plan* composed of *physical operators* which are responsible for executing the query. Query processing is discussed in somewhat more depth in Section 2.5.1, then in great detail in subsequent chapters.

The *data engine* layer houses the *OEM object manager*, *physical operators*, *DataGuide manager*, *external data manager*, *statistics manager*, *index manager*, *lock manager*, and *logging component*. The object manager functions as the translation layer between OEM and 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 object manager also dictates the physical layout of objects on disk. The physical operators are responsible for executing the physical query plan and generating the result to the user's query. Details on the physical operators appear in Chapter 3. Lore's DataGuide manager is responsible for creating and maintaining the DataGuide, which is a dynamic structural summary based on the current data in the database. Some details on the DataGuide appear in Section 2.5.3. The external data manager allows Lore to dynamically fetch and integrate data from external sources at query execution time. The external data manager is discussed in detail in Chapter 8. Statistics for atomic values and the shape of the database graph are provided by the statistics manager. Statistics are discussed in detail in Chapter 3. The index manager provides both value indexing and path indexing capabilities, to allow for fast access to data. Some details are given in Section 2.5.2. Multi-user support in Lore is handled by the lock manager and logging component. The lock manager uses a strict two-phase page-level locking protocol as described in [GR92]. The logging component follows the generic page-level undo/redo logging strategy, also described in [GR92].

2.5.1 Query Processing

A query is submitted through the API in textual form. After parsing, the preprocessor transforms the query by: (1) introducing a *from* clause if it has not been specified (see Example 2.4.7); (2) eliminating Lorel shortcuts (for example "`.#`" becomes "`(.%)*`"); (3) finding common path prefixes and introducing variables (see Example 2.4.2); (4) breaking each path expression into its component form (see Section 2.4.7); (5) introducing explicit existential quantification in the *where* clause when the user has specified unqualified path expressions in a predicate (see Example 2.4.4); and (6) expanding all label wildcards (see Section 2.4.3).

The removal of label wildcards (step 6 above) is accomplished by transforming each label l containing one or more wildcard symbols "%" (recall Section 2.4.3) into an alternation of all possible matching labels in the database. Lore tracks the set of all labels in the database and uses a simple string matching algorithm to determine the set $\{l_1,\ldots,l_n\}$ of possible matching labels. The original label l is replaced by a subpath containing the alternation $(.l_1|\ldots|.l_n)$. For example, the path expression component "`m.Author% a`" is expanded into "`m(.Author|.Authors|.AuthorName) a`" when `Author`, `Authors`, and `AuthorName` are all labels appearing in the database that match `Author%`. When l is already participating in an alternation expression, for example "`m(.Penname|.Author%) a`", then the expansion is inlined within the existing alternation: "`m(.Penname|.Author|.Authors|.AuthorName) a`".

Once steps (1)—(6) above are completed, the transformed parse tree is passed to the logical query plan generator where a single logical query plan is generated in a very straightforward manner. Details on this process appear in Chapter 3. A logical plan specifies the logical steps required to answer a user's query, without being specific about operation ordering or physical access to the data. The query optimizer may rewrite the logical query plan by transforming it into an equivalent logical plan that may generate more efficient physical query plans.

From a logical query plan many physical query plans are considered. A physical query plan is composed of physical operators that are each responsible for some facet of the work required to answer a user's query. Each physical query plan is assigned an estimated "cost" that would be incurred if that plan were executed. The cost of a plan can be measured by many different factors including overall execution time, time to produce the first object in the result, CPU load, etc. In Lore the cost of a physical query plan is the estimated running time of the plan, measured by the estimated number of object fetches. The manner in which the physical query plans are generated and assigned a cost is discussed in more detail in Chapter 3. Lore selects the physical query plan it considers with the smallest estimated cost.

The result of a Lorel query is a set of objects, where each object becomes a subobject of a newly created named object `Answer`. The `Answer` object from the previous query is overwritten, although the user can create a different named object for the query result or can use an update statement to move the previous answer. The oid for the `Answer` object is returned through the API. The application may then use routines provided by the API to traverse the result subobjects and display them in a suitable fashion to the user.

Iterators and Evaluations

Our query execution engine is based on a recursive *iterator* approach. In the iterator model each physical operator supports three operations: *Open*, *GetNext* and *Close*. If a physical operator has children then each operation may contain iterator calls to its children. *Open* and *Close* are used to begin and end a sequence of *GetNext* calls. Most physical operators perform the bulk of their work in *GetNext*. In the general case, each call to *GetNext* can take as an argument a single "tuple" that has been constructed so far. *GetNext* performs the appropriate action and produces either a new tuple or an end flag as output.

The tuples the Lore system operates on are *evaluations*. An evaluation is a vector with one element in the vector for each variable in the query (including any variables introduced during preprocessing). Each vector element in an evaluation contains the oid of the object bound to the variable (if any), and the last label matched to arrive at the object. During query evaluation there is a known mapping between the variables in the query plan and slots (vector elements) in an evaluation. When it is unambiguous, we may use the terms variable and evaluation slot interchangeably.

2.5.2 Indexing

As we will discuss in Chapter 3, it is very important that Lore's optimizer consider the use of indexes when generating a physical query plan. In this section we provide an overview of the indexes available in Lore and how they are implemented.

In a traditional relational DBMS, an index is created on an attribute in order to locate tuples with particular attribute values (or values satisfying particular conditions) quickly. In Lore, such a *value index* alone is not sufficient, since the path to an object is as important as the value of the object. Thus, Lore supports the following indexes:

- The link index, or *Lindex*, provides parent pointers for a given object. The Lindex lookup operation accepts an object, o, and edge label, l, and returns all parents of o reachable via edge l. (That is object o may have one or more incoming edges labeled l and the *Lindex* will return all parents of o for the l edges.) The Lindex is implemented using linear hashing [Lit80].

- The edge index, or *Bindex*, returns all parent/child oid pairs that are connected via an edge with a given label. The Bindex lookup operation accepts an edge label, l, and returns the set of all $\langle p, c\rangle$ pairs where p and c are oids of objects connected via an edge labeled l. The Bindex is also implemented using linear hashing.

- The path index, or *Pindex*, efficiently supports finding all objects at the end of a given path that begins from a name. The Pindex lookup operation accepts a path expression, p, and returns the set of objects that are reachable via p. The Pindex is supported by Lore's DataGuide (discussed below in Section 2.5.3).

- The value index, or *Vindex*, efficiently supports finding objects with atomic values that satisfy a simple predicate. A Vindex lookup operation takes a label l, operator op, and value v. It returns all atomic objects having an incoming edge labeled l and a value satisfying the op v (e.g., < 5). Because Vindexes are useful for *range* (inequality) as well as *point* (equality) queries, they are implemented as B+-trees [Com79].

- The text index, or *Tindex*, efficiently supports finding objects with atomic string values that satisfy a boolean expression. The basic terms of the boolean expression are keywords, satisfied if the string value contains one or more instances of the keyword. The expression can use the standard boolean operators *and*, *or*, and *not*, as well as the information retrieval style operator *near*. The *near* binary operator evaluates to `true` when its operands appear close to each other in a string value. The boolean expression also can contain the *word completion* operator "`%`", which matches any number of characters in a single word and is identical to the word completion operator supported by the SQL *like* construct. The Tindex is supported in the Lore system by inverted lists (words followed by a list of pointers to atomic objects containing that word), implemented using a linear hashing structure [Lit80]. A second version of the Tindex is implemented using *Glimpse*, a publicly available full-text indexing system [Man98].

These indexes enable many different physical query plans. We now describe Vindexes in more detail, specifically how they handle the automatic atomic value coercion performed by Lore.

Value Indexes

Value indexing in Lore requires some novel features due to Lore's non-strict typing system. When comparing two values of different types, Lore always attempts to first coerce the values into comparable types. Currently, our indexing system deals with coercions involving integers, reals, and strings only. Coercion of these three types is summarized by the following three rules:

1. When comparing a string and a real we attempt to coerce the string into a real. If the coercion is successful then the comparison is performed, otherwise it returns `false`.

2. When comparing a string and an integer we attempt to coerce the string into a real. If the coercion is successful then the integer is cast as a real and the comparison is performed, otherwise it returns `false`.

3. When comparing an integer and a real the integer is cast as a real and the comparison is performed.

In order to build and use value indexes that conform to our coercion rules, Lore must maintain three different kinds of Vindexes:

1. *String Vindex*, which contains index entries for all string-based atomic values (`string`, `HTML`, `URL`, etc.).

2. *Real Vindex*, which contains index entries for all numeric-based atomic values (`integer` and `real`).

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

For each label over which a Vindex is created (as specified by the database user or administrator), three separate B+-trees are constructed. When using a Vindex for a comparison (e.g., find all *Age* objects > 30), there are two cases to consider based on the type of the comparison value:

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

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

2.5.3 DataGuides

Without some knowledge of the structure of the underlying database, writing a meaningful Lorel query may be difficult, even when using general path expressions. One may manually browse a database to learn more about its structure, but this approach is unreasonable for very large databases. A *DataGuide* is a concise and accurate summary of the structure of an OEM database, stored itself as an OEM structure. Each path (sequence of labels) in the database is represented exactly once in the DataGuide, and the DataGuide contains no paths that do not appear in the database. In typical situations, the DataGuide is significantly smaller than the original database. Figure 2.8 shows a DataGuide for the Database Group database shown in Figure 2.2.

In Lore, the DataGuide plays a role similar to the schema in a traditional database system. The DataGuide may be queried or browsed, enabling user interfaces or client

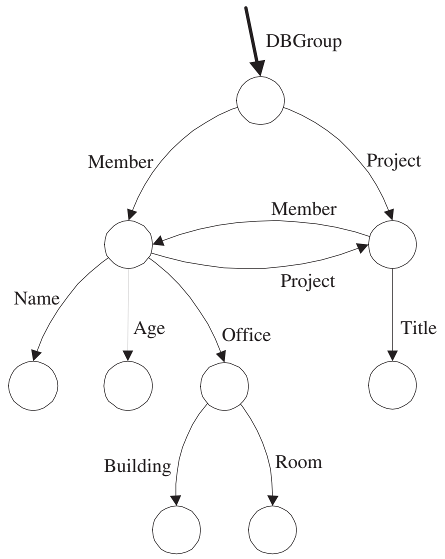

Figure 2.8: A DataGuide for the database in Figure 2.2

applications to examine the structure of the database. The main difference is that in relational or object-oriented systems the schema is explicitly created before any data is loaded, while in Lore DataGuides are dynamically generated and maintained over all or part of an existing database.

DataGuides support storage of *annotations* within objects. An annotation for a set of objects in the database reachable by a given path is stored by assigning it to the single object in the DataGuide reachable by that path. Annotations are useful, e.g., for storing sample atomic values reachable via a given path, or for storing statistics about paths beginning from a named object. Most importantly, we can annotate every object in the DataGuide with a set of database oids in order to support the Pindex.

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

2.5.4 Bulk Loading and Physical Storage

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

Lore arranges objects in physical disk pages; each page has a number of slots with a single object in each slot. Since objects are variable-length, Lore places objects according to a *first-fit* algorithm, and provides an object-forwarding mechanism to handle objects that grow too large for their original page. In addition, Lore supports large objects that may span many pages; such large objects are useful for our multimedia types, as well as for complex objects with many subobjects.

We have a simple object layout scheme. Objects are clustered on a page in a depth-first manner, primarily because one common query execution strategy uses essentially a depth-first search of the data graph (details in Chapter 3). It is obviously not always possible to keep all objects close to their parents (nor is it always desirable), since an object may have several parents. When an object has multiple parents it is stored with an arbitrary parent. Finally, if an object o cannot be reached via a path originating from a named object, then o is deleted by our garbage collector.

2.6 Related Work

Data Model. Lore's data model, OEM, was designed originally for the Tsimmis project at Stanford [PGMW95, PGGMU95, PGMU96]. The Lore project introduced two minor changes to the original OEM data model [PGMW95]: placing labels on edges instead of objects and introducing named objects as a way of beginnning path traversals. *UnQL* [BDS95], introduced in Chapter 1 (Section 1.3), also uses a graph-based data model. All objects in an UnQL graph have a complex value and thus no atomic data values appear in objects. Edges in the graph have three types: *character strings*, *integers*, and *symbols*. Symbols correspond to the labels appearing on edges in OEM, and character strings and integers are similar to OEM's atomic values. In the UnQL data model string and integer data values may occur anywhere in the graph, not just on terminal edges. *Strudel*, also introduced in Chapter 1 (Section 1.3), uses a slight variation on Lore's data model: Strudel uses *collections*, which are sets of nodes, instead of named objects as entry points to a graph.

Query language. A first version of Lorel was introduced in [QRS+95b] and implemented in the initial version of the Lore system. This earlier version of Lorel was designed from scratch, and a full denotational semantics for the language was given in [QRS+95a]. The next version of Lorel [AQM+97], which is the version used in this thesis, is based on the query language OQL. A detailed comparison of the original version of Lorel with more conventional languages such as XSQL [KKS92] and SQL [MS93] appears in [QRS+95b]; most comparisons carry over directly to the version of Lorel presented here.

Another OEM-based query language, *MSL* [PGMU96, PAGM96] (for *Mediator Specification Language*), has been designed in the Stanford Tsimmis project. MSL is a rule-based language with a Datalog [Ull89] rather than OQL or SQL flavor. Its expressiveness is similar to Lorel, however it does not include several features common to database query languages such as aggregation, grouping, and ordering operations.

The UnQL query language [BDS95] contains a powerful construct called *traverse* that allows restructuring of trees to arbitrary depth. Such restructuring operations are not expressible in Lorel, which was designed primarily as a traditional database query language. On the other hand, UnQL lacks several features common to database query languages, including aggregation, grouping, ordering, and update statements.

StruQl [FFLS97], the query language for the Strudel system [FFK+99], is similar to UnQL in that StruQL focuses primarily on object replication and restructuring. StruQL is more powerful than UnQL and Lorel in that it has the same expressive power as first-order logic extended with transitive closure [Imm87]. However, like UnQL, StruQL does not support many standard database constructs.

In [CACS94], a language called *OQL-doc* proposes extensions to OQL that are somewhat similar in spirit to Lorel. Specifically, OQL-doc adds ordering of tuple components and unioning of types to the O_2 data model [BDK92], designed specifically for the management of semistructured SGML data [GR90]. OQL-doc still follows a more rigidly typed approach than Lorel, it supports only the "#" general path expression, and no coercion is performed over atomic values.

The Lore system. The Lore system is unique among the other current projects in managing semistructured data—UnQL [BDHS96], Strudel [FFK+99], and XML-QL [DFF+99] (introduced in Chapter 1, Section 1.3)—since Lore is the only project so far to build a complete, full-featured DBMS. Simple initial implementations of XML-QL [DFF+99] and Strudel [FFK+99] have been created. Both systems are written in Java; the XML-QL implementation consists of 3,610 lines of code and the Strudel system consists of 16,100 lines of code. (By contrast, Lore consists of approximately 168,000 lines of C++ code.) Both systems do not store data under their control. Instead, they accept a query and a local file containing semistructured data. The data is read into main memory and the query is executed, resulting in a new file with the query result. Neither system supports locking, logging, or concurrent access to data.

Chapter 3

Query Optimization Framework

In Chapter 2 we discussed the architecture of the Lore system, including an initial description of the query execution engine and how it fits in relation to other components of the DBMS. In this chapter we present the overall design and implementation of Lore's cost-based query optimizer. The framework presented in this chapter provides a basis for additional, more specific, optimization techniques that we will introduce in Chapters 4, 5, and 6.

Most of the work presented in this chapter appeared originally in [MW99b].

3.1 Introduction

This chapter describes Lore's query processor, with a focus on its cost-based query optimizer. We describe in detail the "logical query plan generator" and "query optimizer" components of the Lore system architecture as shown in Figure 2.7 of Chapter 2.

A cost-based query optimizer in a DBMS is responsible for choosing an efficient *physical query plan* that is then used to evaluate a user's query. The optimizer takes as input the user's query, statistical information about the data, and a set of supported *physical operators* that can be combined to answer the query. The optimizer produces as output a physical query plan that it predicts will be efficient. It is prohibitively expensive for an optimizer to consider all possible physical query plans that could be used to answer a given query, since in general the space of possible plans is very large. Instead, the optimizer considers some subset of all possible plans by removing plans from consideration based on *pruning heuristics* and a general *query plan search strategy*. Each plan that is considered by the optimizer is assigned an estimated *cost* defined by a cost model and cost formulas. The plan with the least estimated cost that is considered by the optimizer is selected for execution. Generally, more time spent optimizing a query results in a better query execution plan, since more plans and optimization techniques can be considered. However, the time spent generating query plans is time taken away from executing a plan (assuming the desired response time is fixed), so the optimizer must balance these considerations carefully.

While our general approach to query optimization is typical—we transform a query into a *logical query plan*, then explore the (exponential) space of possible physical plans looking for the one with least estimated cost—a number of factors associated with semistructured data complicate the problem. Matching path expressions (recall Chapter 2, Section 2.4.1) to paths through the data graph plays a central role in query processing. In Chapter 2 (Section 2.5.2) we introduced a variety of indexes the increase our plan space beyond that of a conventional optimizer, requiring us to develop aggressive pruning heuristics appropriate to our query plan enumeration strategy. Other challenges are to define an appropriate set of statistics for graph-based data, and to devise methods for computing and storing statistics without the benefit of a fixed schema. Statistics describing the "shape" of a data graph are crucial for determining which methods of graph traversal are optimal for a given query and database.

Once we have added appropriate indexes and statistics to our graph-based data model, optimizing the navigational path expressions that form the basis of our query language does resemble the optimization problem for path expressions in object-oriented database systems [GGT96], and even to some extent the join optimization problem in relational database systems [OL90]. As will be seen, many of our basic techniques are adapted from prior work in those areas. However, we decided to build a new overall optimization framework in Lore for a number of reasons:

- Previous work has considered the optimization of single path expressions (e.g., [GGT96, SMY90]). Recall from Chapter 2 (Section 2.4) that our query language permits several, possibly interrelated, path expressions in a single query, along with other query constructs. Our optimizer generates plans for complete queries.

- The statistics maintained by relational DBMS's (for joins) and object-oriented DBMS's (for path expression evaluation) are generally based on single joining pairs or object references, while for accuracy in our environment it is essential to maintain more detailed statistics about complete paths.

- The capabilities of deployed OODBMS optimizers are fairly limited, and we know of no available prototype optimizer flexible enough to meet our needs. Building our own framework has allowed us to experiment with and identify good search strategies and pruning heuristics for our large plan space. It also has allowed us to integrate the optimizer easily and completely into the Lore system.

The remainder of this chapter proceeds as follows. Section 3.2 presents an overview of query processing in Lore. Section 3.3 provides general motivation for query optimization in the context of semistructured data. Section 3.4 presents Lore's query execution engine, including discussion of Lore's logical plans, physical plans, statistics, cost model, and search strategy. Performance results from a range of experiments appear in Section 3.5. Related work is discussed in Section 3.6.

3.2 Lore Query Processing

Recall from Chapter 2 that after a query is parsed, it is first preprocessed to expand Lorel shortcuts and translate the query into a canonical form. The *logical query plan generator* then creates a single logical query plan describing a high-level execution strategy for the query. As we will show in Section 3.4.1, generating logical query plans is fairly straightforward, but special care was taken to ensure that the logical query plans are flexible enough to be transformed easily into a wide variety of different physical query plans. The "query optimizer" component in Figure 2.7 of Chapter 2 actually comprises several steps, as shown in Figure 3.1. Query rewrites are performed over the logical query plan to transform it into an equivalent logical query plan that may improve the optimization process in later stages. The "meat" of the query optimizer occurs in the *physical query plan enumerator*. This component uses statistics and a cost model in order to transform the logical query plan into the estimated best physical plan that lies within our search space. Recall from Chapter 2 that the physical query plan is a tree composed of *physical operators* that are implemented as iterators. The final step to query optimization is the application of *post-optimizations*, which accept as input the single plan generated by the physical query plan enumerator. This final step can fix "mistakes" made by previous steps (due to pruning of the physical search space), or it can introduce new optimizations that could not easily be performed by

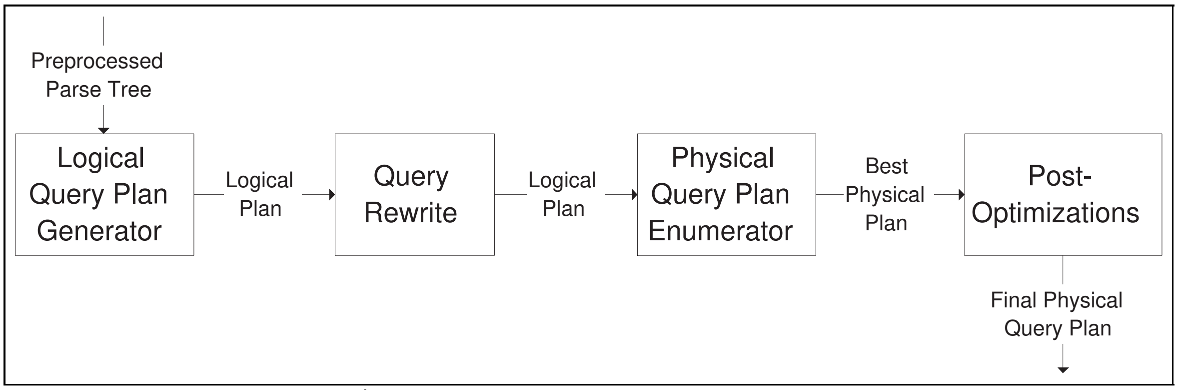

Figure 3.1: The Lore query optimizer

the previous steps.

3.3 Motivation for Query Optimization

As in any declarative query language, there are many ways to execute a single Lorel query. We consider a simple example, Query 3.3.1 introduced below, and roughly sketch several types of query plans. As we will illustrate, the optimal query plan depends not only on the values in the database but also on the *shape* of the graph containing the data. It is this additional factor that makes optimization of queries over semistructured data both important and difficult. The following query is intended for the Database Group database introduced in Chapter 2 (Section 2.3) and shown in Figure 2.2. It finds all group members less than 30 years old.

Query 3.3.1

```
select m
from DBGroup.Member m
where exists a in m.Age: a < 30
```

The most straightforward approach to executing Query 3.3.1 is to fully explore all `Member` subobjects of `DBGroup` and for each one look for the existence of an `Age` subobject of the `Member` object whose value is less than 30. We call this a *top-down* execution strategy since we begin at the named object `DBGroup` (the top), then process each path expression component in a forward manner. This approach is similar to *pointer-chasing* in object-oriented database systems, and to nested-loop index joins in relational database systems.

This query execution strategy results in a depth-first traversal of the graph following edges that appear in the query's path expressions.

Another way to execute Query 3.3.1 is to first identify all objects that satisfy the "`a < 30`" predicate by using an appropriate Vindex if it exists (recall Chapter 2, Section 2.5.2). Once we have an object satisfying the predicate, we traverse backwards through the data, going from child to parent, matching path expressions in reverse using the Lindex. We call this query execution strategy *bottom-up* since we first identify atomic objects and then attempt to work back up to a named object. This approach is similar to *reverse pointer-chasing* in object-oriented systems. The advantage of this approach is that we start with objects guaranteed to satisfy the *where* predicate, and do not needlessly explore paths through the data only to find that the final value does not satisfy the predicate. Bottom-up is not always better than top-down, however, since there could be very few paths satisfying the path expression but many objects satisfying the predicate.

A third strategy is to evaluate some, but not all, of a path expression top-down and create a temporary result of satisfying objects. Then use the Vindex as described earlier and traverse up, via the Lindex, to the same point as the top-down exploration. A join between the two temporary results yields complete satisfying paths. (In fact certain join types do not require temporary results at all.) We call this approach a *hybrid* plan, since it operates both top-down and bottom-up, meeting in the middle of a path expression. This type of plan can be optimal when the fan-in degree of the reverse evaluation of a path expression becomes very large at about the same time that the fan-out degree in the forward evaluation of the path expression becomes very large.

These three approaches give a flavor of the very different types of plans that could be used to evaluate a simple query, one that effectively consists of a single path expression. The actual search space of plans for this simple query is much larger, as we will illustrate in Section 3.4.4, and more complicated queries with multiple interrelated path expressions naturally have an even larger variety of candidate plans.

To make things even more concrete, suppose we are processing the query "`Select b From A.B b Where exists c in b.C: c = 5`", which is isomorphic to Query 3.3.1. In Figure 3.2 we present the general shape and a few atomic values for three databases, illustrating cases when each type of query plan described above would be a good strategy. The database on the left has only one "`A.B.C`" path and top-down execution would explore only this path. Bottom-up execution, however, would visit all the leaf objects with value

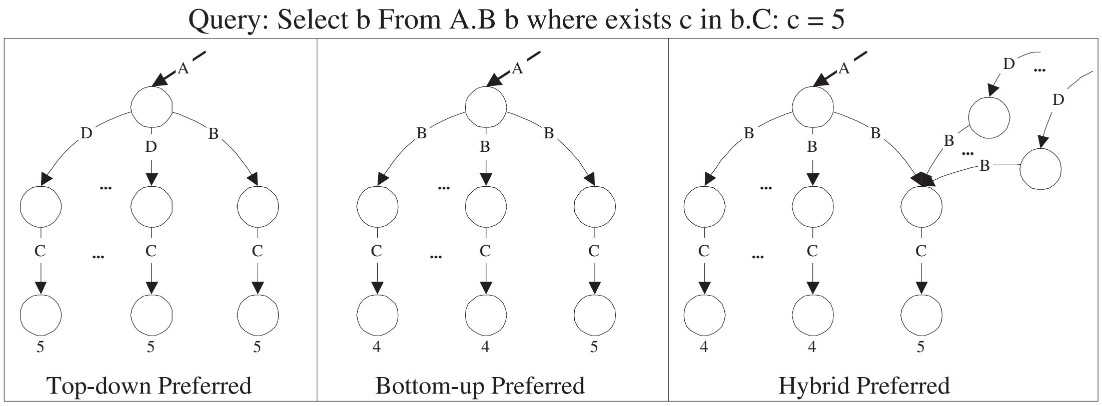

Figure 3.2: Different databases and good query execution strategies

5, and their parents. The second database has many "`A.B.C`" paths, but only a single leaf satisfying the predicate, so bottom-up is a good candidate. Finally, in the third database top-down execution would visit all the leaf nodes, but only a single one satisfies the predicate. Bottom-up would identify the single object satisfying the predicate, but would needlessly visit all of the nodes in the upper-right portion of the database. For this database, a hybrid plan where we use top-down execution to find all "`A.B`" objects, then bottom-up execution for one level, then finally join the two results, would be a good strategy.

Consider how very different just these three example plans are. In top-down we have a forward evaluation of all path expressions in the *from* clause and then evaluation of the *where* clause. In bottom-up, we first handle the *where* clause and then perform a reverse evaluation of all path expressions. The hybrid approach can evaluate either the *from* or the *where* first, but a join must be performed between the two subplans. Each of these three example plans is the optimal plan for a particular database. Our primary goal when designing our logical query plans was to create a structure that represents, at a high level, the sequence of steps necessary to execute a query, while at the same time permits simple rules to transform the logical query plan into a wide variety of different physical query plans.

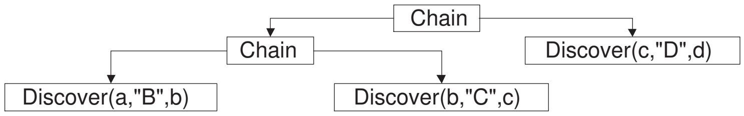

Figure 3.3: Representation of a path expression in the logical query plan

3.4 Query Execution Engine

3.4.1 Logical Query Plans

One major difference between the top-down and bottom-up query execution strategies introduced in Section 3.3 is the order in which different parts of the query are processed. In the top-down approach we handle the *from* clause before the *where* clause; the order is reversed for the bottom-up strategy. Also consider the *where* clause of Query 3.3.1: "`Where exists a in m.Age:a < 30`". We can break this clause into two distinct pieces: (a) find all `Age` subobjects of m, and (b) test their values. In the bottom-up plan we first use the Vindex to satisfy (b) and then we use the Lindex for (a). In the top-down strategy first we satisfy (a) by finding an `Age` subobject of m, then we test the condition to fulfill (b).

In fact, all queries can be broken into independent components where the execution order of the components is not fixed in advance. We term the places where independent components meet *rotation points*, since during the creation of the physical query plan we can rotate the order between two independent components to get vastly different physical plans.

In Table 3.1, we summarize the logical query plan operators used in Lore. We begin describing the logical operators by focusing on the *Discover* and *Chain* logical operators used for path expressions. Each path expression component in the query (recall the definition of path expression component from Chapter 2, Section 2.4.7) is represented as a *Discover* node, which indicates that in some fashion information is discovered from the database. When multiple path expression components are grouped together into a path expression, we represent the group as a left-deep tree of *Discover* nodes connected via *Chain* nodes. It is the responsibility of the *Chain* operator to optimize the entire path expression represented in its left and right subplans. As an example, consider the path expression "`a.B b, b.C c, c.D d`" (where a is defined elsewhere in the query) which has the logical query subplan shown in Figure 3.3. The left-most *Discover* node is responsible for choosing the best way to provide bindings for variables a and b. The *Chain* node directly above it is

Node	Parameters	Description
Project	Variable	Project *Variable*.
Select	Predicate	Apply the *Predicate* to the current evaluation.
CreateTemp	PrimVar, SecVar, DestVar	Accumulate a set of intermediate structures placed in the destination variable, *DestVar*, where each intermediate structure tracks a primary variable, *PrimVar*, and a set of secondary variables, *SecVar*.
Set	LeftPlan, RightPlan, SetOp	Perform a set operation *SetOp* (*union*, *intersect*, or *except*) using two sets of CreateTemp structures passed up from the children nodes.
Glue	LeftPlan, RightPlan	Used to connect two independent subplans, where in general either subplan could be executed first.
Name	Name, DestVar	Find the named object, *Name*, and place the object into the destination variable, *DestVar*.
Chain	LeftPlan, RightPlan	Connector used between two components of a path expression.
Discover	Path Expression Component	Used to discover information from the database based on the path expression component.
Exists	Variable	Ensure that there is a binding for *Variable*.
Compound	LeftPlan, RightPlan, CoOp	Links two components of a compound predicate, where the operator *CoOp* can be either *and* or *or*.
Update	LeftPlan, RightPlan, UpdateOp	Perform the *UpdateOp*, altering the objects returned by the *LeftPlan* based on results from the *RightPlan*.
With	SourceVar, DestVar, TargetVar	Replicate the *DestVar*, which was discovered from *SourceVar*, into *TargetVar*. Used for materialized views (Chapter 7).
Aggr	SourceVar, AggrOp	Apply aggregation operation *AggrOp* to objects bound to *SourceVar*.
Arith	LhsVar, RhsVar, ArithOp	Apply arithmetic operation *ArithOp* to objects bound to *LhsVar* and *RhsVar*.

Table 3.1: Logical query plan operators

responsible for evaluating the path expression "`a.B b, b.C c`" efficiently. This evaluation could be done by using the children's most efficient ways of executing their subplans and joining them together in some fashion, or possibly by using a path index for the entire path expression. The final *Chain* and *Discover* nodes are similar.

As described in Chapter 2 (Section 2.4.1), a name is a reference to a single object and is used as an entry point into the database. We use a special logical query plan operator *Name* to discover information about named objects. Names have several special properties including the fact that they are unique, and they are stored efficiently due to their frequent access. Also, since the database is divided into *workspaces* (to handle views and private areas within which individual users can work), the *Name* operator is a convenient place to record the workspace over which a path expression is being evaluated.

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

The *CreateTemp* operator logically creates a set of intermediate structures based upon the evaluations of its children. It is used by the set operations (*union*, *intersect*, *except*) and the physical *HashJoin* operator (described in Section 3.4.2), all of which operate over sets of evaluations, and is similar to creating a temporary table in a relational query plan. One variable, bound by the subplan of the *CreateTemp* operator, is designated as *Primary*. A list of variables, which also must be bound by a descendant, is designated as *Secondary*. Each intermediate structure associates, for one object bound to the primary variable, a set of bindings for each secondary variable. Since the structure may not fit in memory, care is taken in storing the structure efficiently. In Section 3.4.2 we will explain why the *CreateTemp* operator creates secondary variable bindings and not just a straightforward collection of evaluations.

For a brief description of the remaining logical operators (*Set*, *Glue*, *Update*, *With*, *Aggr*, and *Arith*) please see Table 3.1.

Figure 3.4 shows the complete logical query plan for Query 3.3.1. Each *Glue* and *Chain* node is a rotation point, which has as its children the two independent subplans. The topmost *Glue* node connects the subplans for the *from* and *where* clauses. The *Chain* node

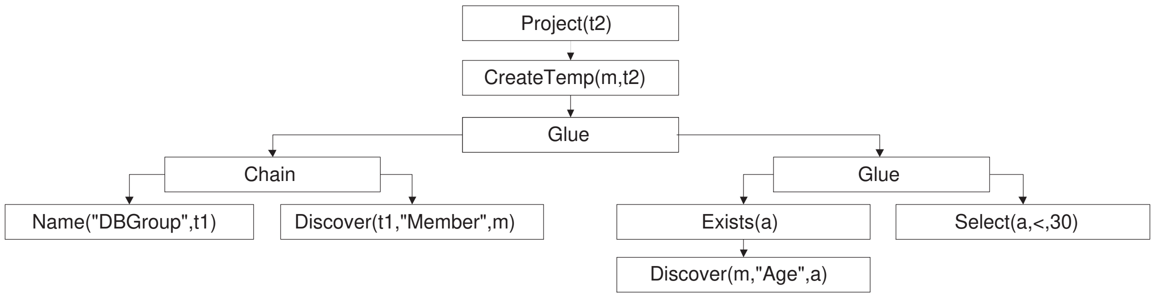

Figure 3.4: A complete logical query plan

connects the two components of the path expression appearing in the *from* clause. The *Exists* node quantifies a. A *Glue* node separates the existential in the *where* from the actual predicate test, allowing either operation to occur first in the physical query plan. Because the semantics of Lorel requires a set of objects to be returned, the *CreateTemp* and *Project* nodes at the top of the plan are responsible for gathering the satisfying evaluations and returning the appropriate objects to the user.

3.4.2 Physical Query Plans

Lore's physical operators are summarized in Tables 3.2 and 3.3. Recall from Chapter 2 (Section 2.5.1) that our physical query plan operators are implemented as iterators. Each iterator supports three methods: *Open*, *GetNext*, and *Close*. An evaluation is supplied as input to each call of *GetNext* and a resulting evaluation is returned. Recall (also from Chapter 2, Section 2.5.1) that an evaluation is a vector with one element in the vector for each variable in the query and each temporary variable. In the *GetNext* method, most physical operators access some objects bound to variables in the evaluation passed in, perform some work, and bind additional variables in the evaluation returned. The returned evaluation is passed up to the parent operator, and the overall goal of physical query plan execution is to pass complete evaluations to the root of the physical query plan.

In Figures 3.2 and 3.3, and also in the explanation of the physical operators that appears below, we list the parameters of each of the physical operators. In these descriptions x, y, *PrimVar*, *DestVar*, *TargetVar*, *SourceVar*, *LhsVar*, *RhsVar*, and *Variable* are variables in the query (or temporary variables) and thus can be mapped directly to a slot in the evaluation. In addition: *SecVar* is a set of variables; l is a label; *LeftPlan* and *RightPlan* are the two

Name	Parameters	Function
Project	Variable	Project out *Variable*.
Select	Predicate	Apply the *Predicate* to the evaluation.
NLJ	LeftPlan, RightPlan, Predicate	Relational nested-loop join.
HashJoin	LeftPlan, RightPlan, Predicate	Relational hash join.
SMJ	LeftPlan, RightPlan, Predicate	Relational sort-merge join.
Sort	PrimVar, SecVar	Sort the evaluations passed from child based on oid of *PrimVar*. Also remembers bindings for all *SecVar*.
Compound	LeftPlan, RightPlan CoOp	Link two components of a compound predicate,where the operator, *CoOp*, can be either *and* or *or*.
Scan	x, *Subpath*, y	Place all descendants of x that are reachable via *Subpath* into y.
Lindex	x, *Subpath*, y	Place all ancestors of the object in y reachable via *Subpath* into x.
Pindex	PathExpression, DestVar	Using the DataGuide's path index, put all objects reachable via *PathExpression* into *DestVar*.
Bindex	x, l, y	Find all edges with label l and place the parent into x and the child into y.
Vindex	Label, Op, Value, DestVar	Find all atomic values satisfying *Op Value* and that have an incoming edge labeled *Label* and place them into *DestVar*.
Tindex	Expr, DestVar	Find all atomic string values satisfying the boolean expression*Expr* and place them into *DestVar*.
Name	SourceVar, Name	Confirm that the object in *SourceVar* is the named object *Name*.
Once	Variable	Ensure that each object is only assigned to *Variable* a single time.
CreateTemp	PrimVar, SecVar, DestVar	Accumulate a set of intermediate structures, where each intermediate structure tracks information about a primary variable, *PrimVar*, and a set of secondary variables, *SecVar*.

Table 3.2: Physical query plan operators

Name	Parameters	Function
Set	LeftPlan, RightPlan, SetOp	Perform a set operation *SetOp* (*union*, *intersect*, or *except*) using two sets of CreateTemp structures passed up from the children nodes.
Deconstruct	SourceVar	Take the intermediate structure in *SourceVar* and decompose it into its components.
ForEach	SourceVar	Take the set of objects in *SourceVar* and iterate over them one at a time.
Aggregation	SourceVar, Op, DestVar	Perform the aggregation operation, *Op*, over *SourceVar*. In addition, this operator can be used to ensure that at least one object was successfully bound to *SourceVar*.
Update	LeftPlan, RightPlan, UpdateOp	Perform the *UpdateOp* altering the objects returned by the *LeftPlan* by results from the *RightPlan*.
Oem	Label, Value, DestVar	Creates a new object with value *Value* and places it into *DestVar*. When *Label* is specified then also creates the name *Label* to the newly created object.
With	SourceVar, DestVar, TargetVar	Replicate the *DestVar*, which was discovered from *SourceVar*, into *TargetVar*. Used for materialized views (Chapter 7).
Arith	LhsVar, RhsVar, ArithOp	Apply arithmetic operation *ArithOp* to objects bound to *LhsVar* and *RhsVar*.

Table 3.3: More physical query plan operators

subplans for a non-leaf binary physical operator; *Predicate* is a simple predicate of the form x op y, where x and y can be either variables or values.

Operators to Discover Information

In a physical query plan, there are seven operators that identify information stored in the database:

1. *Scan*(x, *Subpath*, y): The *Scan* operator performs pointer-chasing: it places into y all objects that are descendants of the complex object x via the subpath *Subpath* (recall Chapter 2, Section 2.4.7).

2. *Lindex*(x, *Subpath*, y): In the reverse of the *Scan* operator, the *Lindex* operator places into x all objects that are ancestors of y via the subpath *Subpath*. See Chapter 2 (Section 2.5.2) for a description of the Lindex itself.

3. *Pindex*(*PathExpression*, x): The *Pindex* operator places into x all objects reachable via the *PathExpression*. See Chapter 2 (Section 2.5.3) for a description of the Pindex itself.

4. *Bindex*(l, x, y): The *Bindex* operator finds all parent-child pairs connected via an edge labeled l, placing them into x and y. This operator allows us to efficiently locate edges whose label appears infrequently in the database. See Chapter 2 (Section 2.5.2) for a description of the Bindex itself.

5. *Name*(x, n): The *Name* operator simply verifies that the object in variable x is the named object n.

6. *Vindex*(l, *Op*, *Value*, x): The *Vindex* operator accepts a label l, an operator *Op*, and a *Value*, and places into x all atomic objects that satisfy the "*Op Value*" condition and have an incoming edge labeled l.

7. *Tindex*(*Expr*, *DestVar*): The *Tindex* operator accepts a boolean expression *Expr* and places into *DestVar* all atomic string objects that satisfy *Expr*. See Chapter 2 (Section 2.5.2) for a description of the allowed boolean expressions and the Tindex itself.

As an example that uses some of these operators, consider the path expression "`A.B b, b.C c`" (where `A` is a name). This path expression becomes three path expression components: $\langle$ `Root`, `A`, $a\rangle$, $\langle a$, `B`, $b\rangle$, $\langle b$, `C`, $c\rangle$. Four possible physical plans are shown in Figure 3.5.

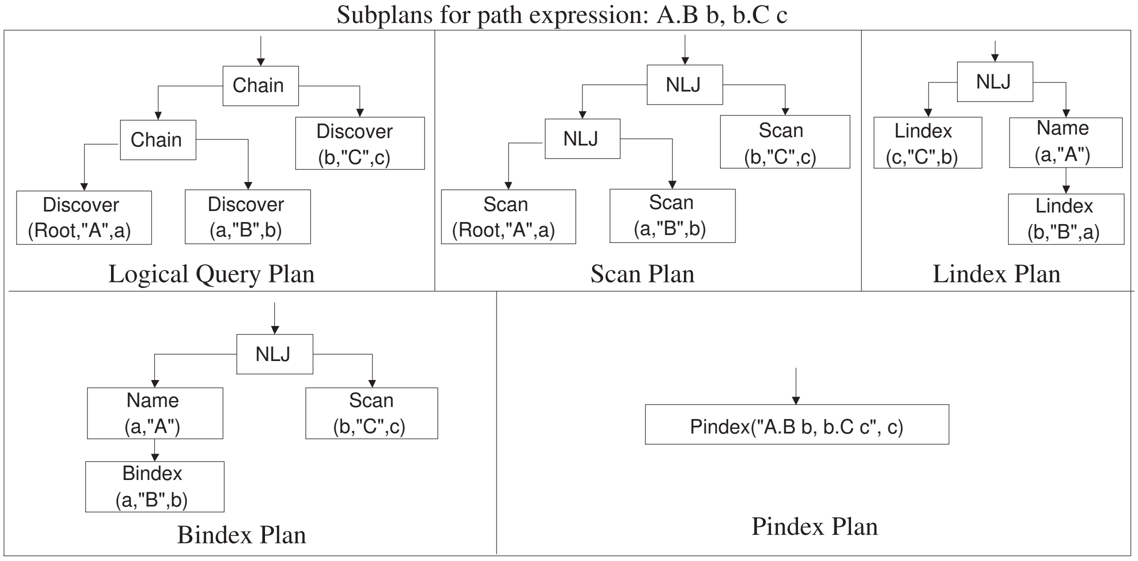

Figure 3.5: Different physical query plans

(The optimizer can generate up to eleven different physical plans for this single path expression.) The logical query plan is shown in the top left panel. In the first physical plan, the "Scan Plan", we use a sequence of *Scan* operators to discover bindings for each of the path expression components which corresponds to the top-down execution strategy introduced in Section 3.3. If we already have a binding for c then we can use the second plan, the "Lindex Plan". In this plan we use two *Lindex* operations starting from the bound variable c, and then confirm that we have reached the named object `A`. This corresponds to the bottom-up execution strategy of Section 3.3. In the "Bindex Plan", we directly locate all parent-child pairs connected via a `B` edge using the *Bindex* operator. We then confirm that the parent object is the named object `A`, and *Scan* for all of the `C` subobjects of the child object. In the "Pindex Plan", we use the *Pindex* operator, which allows us to directly obtain the set of objects reached via the given path expression. Note that several of the plans use a nested-loop join (NLJ) operator without a predicate. These are *dependent joins* where the left subplan passes bound variables to the right subplan [CD92].

Recall the hybrid query execution strategy introduced in Section 3.3. One subplan evaluates a portion of the query and obtains bindings for a set of variables, say V, and another subplan obtains bindings for another set of variables, say W. Suppose $V \cap W$ contains one variable, but the plans are otherwise *independent*, meaning one does not provide a binding that the other uses (as in the hybrid plan). Then by creating evaluations for both subplans and joining the results on the shared variable, we efficiently obtain complete evaluations. As in relational systems, deciding which join operator to use is an important decision made by the optimizer. Lore supports nested-loop, hash, and sort-merge joins [Ull89].

Regular Expression Operators

The *Scan* and *Lindex* operators provide bindings for a variable based on a given *Subpath*. As described in Chapter 2 (Section 2.4.3), a subpath may contain either a single label, the alternation of a set of labels, or either of these types followed by ?, *, or +. (Recall that we have not implemented the full generality of regular path expressions.) Both the *Scan* and *Lindex* operators use an internal stack of objects in order to support the evaluation of subpaths with ?, *, or +. We describe how the *Scan* operator works; *Lindex* is similar except we look for ancestors of y instead of descendants of x. Consider the path expression component $\langle x$,(L_1 | L_2 | ... | $\mathrm{L}_n)^{op}$, $y\rangle$ where *op* is ?, *, or +. To begin, the object bound to source variable x is pushed onto the stack. If *op* is * or ? then the object bound to x is bound to y and the evaluation is passed to the *Scan*'s parent. Otherwise (and also on each subsequent call to *GetNext* regardless of *op*) the stack is popped and (L_1 | L_2 | ... | L_n) is evaluated with respect to the object that was at the top of the stack. Every object resulting from the evaluation of the subpath is bound to y. If *op* is * or + then each object bound to y is also pushed onto the stack. Traversing a cycle multiple times is avoided by only pushing objects onto the stack if they do not already appear there. The execution is complete for a sequence of *GetNext* requests to the *Scan* operator when the stack is empty.

Note that this execution strategy can be very inefficient. For example, suppose the following path expression is evaluated over the Database Group database: "`DBGroup.Member m, m.# x, x.Project p`". Recall from Chapter 2 (Section 2.4.3) that "#" is shorthand for (.%)*, i.e., # matches zero or more edges with any label. In a top-down query execution strategy a *Scan* operator is used to produce bindings for x in path expression component $\langle m$, #, $x\rangle$. Every descendant of m will be bound to x, and if m has numerous descendants but few of them lead to projects then significant needless exploration of the database is performed. We consider an alternative approach to handling general path expressions in Chapter 4.

Evaluations			Encapsulated Evaluation Set
x: &1	y: &3	z: &7	x: &1, t: { ⟨y:&3, z:&7⟩,
x: &1	y: &3	z: &8	⟨y:&3, z:&8⟩,
x: &1	y: &4	z: &7	⟨y:&4, z:&7⟩ }
x: &2	y: &6	z: &9	x: &2, { ⟨y:&6, z:&9⟩ }

Table 3.4: Example of an Encapsulated Evaluation Set

Encapsulating a Set of Evaluations

Some physical operators in Lore operate over a set of evaluations produced by a subplan, rather than over one evaluation at a time. For efficiency, we define an *encapsulated evaluation set* (EES) to be an internal representation that transforms a set of evaluations of the form $\langle v_1 : o_1, v_2 : o_2, ..., v_n : o_n \rangle$ into a (potentially smaller) set of evaluations of the form $\langle v_1 : o_1, v_g : \{\langle v_2 : o_2, ..., v_n : o_n\rangle\}\rangle$. Variable v_1 is called the *primary variable* and variables v_2 through v_n are the set of *secondary variables* for the EES. Variable v_g is a new variable holding the complete set of evaluations for the secondary variables for a given object bound to the primary variable.

As an example, consider the four evaluations on the left side of Table 3.4. These evaluations can be represented in an EES containing only two evaluations (shown on the right side of Table 3.4) by using variable x as the primary variable and variables y and z as secondary variables. Each distinct object binding of x appears a single time with the set of associated $\langle y, z \rangle$ binding pairs.

Secondary variables in an EES need to be "unnested" for those physical operators requiring one evaluation at a time. We describe in this chapter one way to revert an EES back to its original form, and describe another way in Chapter 6.

CreateTemp, Deconstruct, and ForEach Physical Operators

The *CreateTemp* operator calls its subplan to exhaustion and creates an EES stored on disk representing all the evaluations passed up by its child. *CreateTemp* further "wraps" the EES into a single evaluation passed to its parent. *CreateTemp* is used to encapsulate the result of a query or subquery, before a set operation (the *Set* operator), and before a hash-join (the *HashJoin* operator).

A structure created by the *CreateTemp* operator must be "deconstructed" and then "flattened" (via physical operators *Deconstruct* and *ForEach*) when query execution needs to operate over the evaluations in the structure. The *Deconstruct* physical operator accepts a *SourceVar* that holds the oid of the single evaluation created by a *CreateTemp* operation, and it decomposes the structure into the evaluations contained within the EES by placing, one at a time, the primary objects into the original variable slot and the structured result of the secondary variables into the new variable created for the EES. After deconstruction, secondary variables are still encapsulated within the structure in the form $\{\langle v_2{:}o_2, ..., v_n{:}o_n\rangle\}$. It is the responsibility of the *ForEach* operator to iterate through the internal representation and place each secondary variable into its original variable slot. The following example illustrates the use of *Deconstruct* and *ForEach*.

Example 3.4.1 Suppose the following query is executed over the Database Group database (Chapter 2, Figure 2.2):

```
select f1, f2
from DBGroup.Member m, m.Advisor a, m.Favorites f1, a.Favorites f2
```

One possible query plan is shown in Figure 3.6. The *CreateTemp* operators are used to feed the *HashJoin* operator. The resulting structure from the *HashJoin* operator is the same as the *CreateTemp* operator and contains an EES with variable m as the primary variable and a as the secondary variable. After the *HashJoin* a *Deconstruct* is used to "unnest" and provide access to variable m. Query processing continues with variable m (performing a scan for "m.favorites") until variable a is required. Then variable t, the temporary variable created by the EES for the secondary variable structures, must be flattened to provide access to variable a. Note that deferring the *ForEach* operator until after the *Scan* for "`m.Favorites f1`" allows us to find bindings for variable $f1$ for each m and not for each $\langle m,a\rangle$ binding. □

Remaining Physical Operators

We now describe the functionality of the physical operators in Table 3.2 that we haven't yet covered.

The *Set* physical operator performs any of the standard set operations, *union*, *intersect*, or *except*, over the two temporary results passed up from its children. The result from the set operation is another temporary structure. The primary variables of the incoming structures

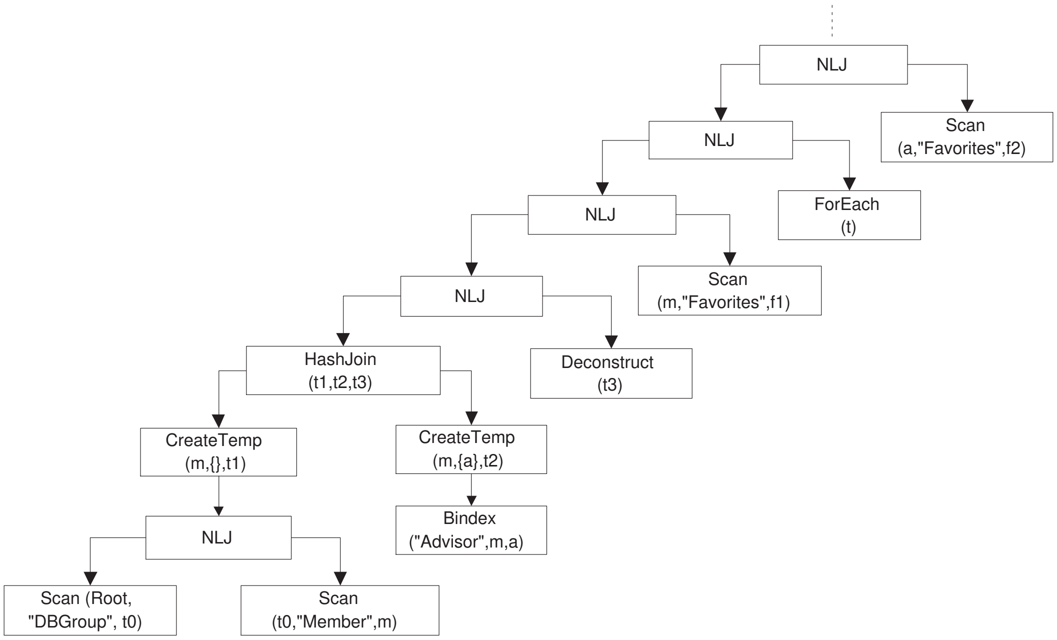

Figure 3.6: Sample physical plan with Deconstruct and ForEach operators

are used to perform the actual set operation, while the resulting secondary structure is the union of the two secondary structures for the satisfying primary objects.

The *Arith* physical operator performs any of the standard arithmetic operations, `+`, `-`, `/`, and `*`. These operations are performed over the results from a left and right subplan. In general, if a set of results are returned from the subplans of the *Arith* operator then the arithmetic operation is performed for each pair of elements in the sets, i.e., over the cross product.

The *Aggregation* physical operator can be used both for standard aggregation operations and also to ensure that a variable satisfies an existential quantification. Aggregation with one of the five standard aggregation operations (*min*, *max*, *sum*, *count*, and *average*) executes by requesting all tuples from the *Aggregation*'s child node and performing the aggregation operation incrementally. *Aggregation* with the *exists* operation calls its subplan until a single binding has been found for its *SourceVar*. Subsequent calls to the aggregation operation with the same evaluation as the previous call will result in no evaluation being passed up, since the existential clause has already been satisfied.

The *Once* operator also is used to ensure that a variable is existentially quantified, but *Once* is used during bottom-up evaluation and appears directly above the *Lindex* node that binds the existentially quantified variable. The *Once* operator only allows an evaluation to be passed to its parent node if the object bound in *Variable* has never been seen before.

The *Update* physical operator appears as the root of a physical query plan for an update statement. The *Update* operator performs either an edge creation, edge deletion, edge replacement, or atomic value update, as specified by the update statement (recall Chapter 2, Section 2.4.4). We describe the *Update* physical operator in more detail in Section 3.4.5. Briefly, the *Update* physical operator is a binary operator where the left subplan locates objects whose value are to be modified and the right subplan locates or creates the new values.

The *Oem* physical operator creates a new OEM object with value *Value* and binds the new object to the variable *DestVar*. The *Oem* physical operator most often appears as the right child of an *Update* operator, however it can also appear as the root of a query plan, or within a subplan for the *select* clause of a *select-from-where* query. *Oem* appears as the root of a query plan when it is being used to create or destroy a named object. In these situations *Label* is the named object, and deletion is indicated by a special flag in the *Value* parameter.

The *Compound* physical operator is used by compound predicates appearing in the *where* clause. The *LeftPlan* and *RightPlan* correspond to the physical query plans used to evaluate the two parts of the compound operation. The operator can be `and` or `or`, with their standard boolean semantics applied to the results of the subplans. The *Compound* operator supports short-circuiting in the usual manner.

The *Sort* operator sorts the set of evaluations passed up by its subplan based on the oid of the *PrimVar*. This operator will create a temporary result if the sorted set does not fit into the main memory segment allocated to it. A standard multi-pass sort as described in [Gra93] is used. This operator is used in conjunction with the *SMJ* operator to support standard relational sort-merge join. (The *SMJ* operator actually performs only the merge step between two sorted relations.) Other query optimization techniques introduced in subsequent chapters also use the *Sort* operator.

The *With* operator is used to replicate objects when they are placed in a materialized view. In materialized views (see Chapter 7) a mapping is maintained between original objects and view objects. The object bound to *DestVar* is replicated and placed into *TargetVar*. The newly created object in *TargetVar* may need to become a child of a previously replicated object. The *SourceVar* (if one exists) identifies the original object that maps to the view object p that should become the parent. An edge is created between p and *TargetVar*'s object.

The *Project* operator is used to remove bindings for all variables except the projected variable. The *Select* operator is used to apply a selection condition to evaluations. When the *Select* operator has a subplan then the subplan is executed and the selection is applied to each evaluation. Those evaluations that pass the selection condition are passed up to the *Select*'s parent operator. If *Select* does not have a subplan then the selection condition is applied to the evaluation passed down from the *Select*'s parent. If the selection condition returns *true* then the evaluation is passed back up to the parent operator. If the selection returns *false* then an end condition is passed to the *Select*'s parent.

In Figure 3.7 we give three complete physical query plans for Query 3.3.1. Each plan corresponds to one of the strategies that we introduced in Section 3.3. We emphasize that these are only three representative plans, while numerous others are possible and are considered by our plan enumerator. In fact, the optimizer can produce 32 different plans for this simple query.

3.4.3 Statistics and Cost Model

As with any cost-based query optimizer, we need to establish a metric by which we estimate the execution cost for a given physical query plan or subplan. The estimated cost for a physical query plan is typically divided into two parts: the *CPU cost* and the *I/O cost*. The CPU cost is an estimate of how much work is required by the processor to execute the plan, usually estimated in number of operations. The I/O cost is an estimate of how much activity will occur between processor and disk, usually estimated as number of page reads and writes. Since I/O operations are typically much more expensive than CPU operations, some commercial DBMSs only estimate the I/O cost of a plan. However, other systems determine the speed of the processor and storage medium and combine CPU and I/O costs into a single number representing overall cost.

The Lore system estimates both CPU and I/O costs of a query plan. Most commercial DBMSs are able to estimate the number of page reads and writes as their I/O cost. However, Lore does not enforce any object clustering, so we cannot accurately determine whether two objects will be on the same page. Thus, we are limited to using the predicted number of object fetches as our measure of I/O cost. Despite this limitation, experiments presented in Section 3.5 validate that our cost model is reasonably accurate.

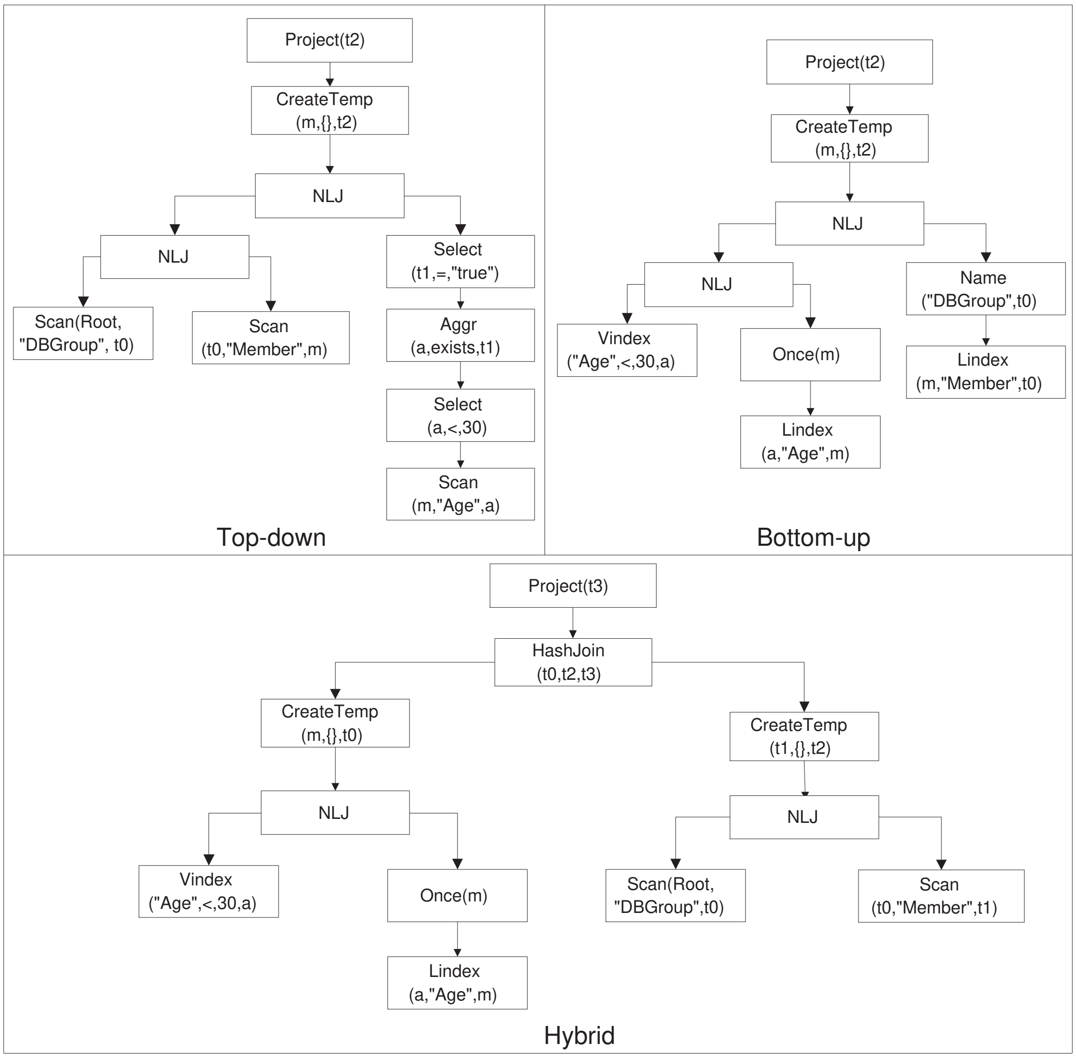

Figure 3.7: Three complete physical query plans

Statistics

Our query optimizer must consult statistical information about the size, shape, and range of values within a database in order to estimate the cost of a physical query plan. Initially we stored statistics in the DataGuide, but quickly were limited by the fact that we could only store statistics about paths beginning from a named object.

Traditional relational and object-oriented statistics are well-suited for estimating predicate selectivities, and for estimating the number of tuples one relation (or class) produces when joined with another relation (or class). (Object-oriented statistics can be somewhat more complicated if the class hierarchy is taken into account, e.g., [CCY94, RK95, SS94, XH94].) However, these statistics are not well-suited for long sequences of joins as embodied in path expressions. A cost-based optimizer for path expressions may, for example, need to accurately estimate the number of "`DBGroup.Member.Advisor.Paper`" paths in the database. In Lore we set a threshold k, and gather statistics for all label sequences (paths) in the database up to length k. (Typical object-oriented and relational database systems compute and store statistics for $k = 1$.) Obviously for large k the cost of producing the statistics can be quite high, especially for cyclic data. A clear trade-off exists between the cost in computation time and storage space for a larger k, and the accuracy of the statistics.

The statistics we maintain, for every label sequence p of length $\leq k$ appearing in the database, include:

- For each atomic type, the number of atomic objects of that type reachable via p.
- For each atomic type, the minimum and maximum values of all atomic objects of that type reachable via p.
- The total number of instances of path p, denoted $|p|$.
- The total number of distinct objects ending in path p, denoted $|p|_d$.
- The total number of distinct objects beginning a path p, denoted $|p|^d$.
- For each label l in the database, the total number of l-labeled subobjects of any object reachable via p, denoted $|p_l|$.
- For each label l in the database, the total number of incoming l-labeled edges to any instance of p, denoted $|p^l|$.

As mentioned earlier, our I/O cost metric is based on the estimated number of objects fetched during evaluation of the query. For example, given an evaluation that corresponds to a traversal to some point in the data, the optimizer must estimate how many objects will bind to the next path expression component to be evaluated. Consider evaluating the path expression "`A.B b, b.C c`" top-down. If we have a binding for b, then the optimizer needs to estimate the number of `C` subobjects, on average, that objects reached by the path "`A.B b`" have. Alternatively, if we proceed bottom-up with a binding for b, then the optimizer must estimate the average number of parents via a `B` edge for all the `C`'s. We call these two estimates *fan-out* and *fan-in* respectively. The fan-out for a given path expression p and label l is computed from the statistics by $|p| * (|p_l|/|p|_d)$. Likewise, fan-in is $|p| * (|p^l|/|p|_d)$.

Our statistics are most accurate for path expressions of length $\leq k+1$, since for a given k we store statistics about paths of length up to k, and these statistics include information about incoming and outgoing edges to the paths—effectively giving us information about all paths of length $k+1$. Given a path expression of length $k+2$, for maximum accuracy we combine the statistics for two overlapping paths p_1 and p_2 each of length $k+1$. We combine the statistics of the two paths using the formula $|p| = |p_2| * |p_1|/|p_1 \cap p_2|$, where $p_1 \cap p_2$ is a third path expression containing all components common to p_1 and p_2.

When estimating the number of atomic values that will satisfy a given predicate, standard formulas such as those given in [SAC$^+$79, PSC84] are insufficient in our semistructured environment due to the extensive type coercion that Lore performs (recall Chapter 2). Our new formulas, given below, take coercion into account by combining value distributions for all atomic types that can be coerced into a type comparable to the value in the predicate.

Cost Model

Each physical query plan is assigned a *cost* based upon the estimated I/O and CPU time required to execute the plan. The costing procedure is recursive: the cost assigned to a node in the query plan depends on the costs assigned to its subplans, along with the cost for executing the node itself. In order to compute estimated cost recursively, at each node we must also estimate the number of evaluations expected for that subplan. To decide if one plan is cheaper than another, we first check the estimated I/O cost. Only when the I/O costs are identical do we take estimated CPU cost into account. Our cost metric, while admittedly simplistic, appears acceptable as shown by the performance results in

Section 3.5.

Our formulas for estimating I/O cost, CPU cost, and number of expected results are given in Tables 3.5, 3.6, and 3.7, respectively. The CPU cost formulas take into account the major CPU operations required to execute a subplan. Since the CPU costs are used only as a tiebreaker when two query plans have exactly the same estimated I/O cost, we wanted to capture the relative amount of CPU work required for each operator without getting bogged down with intricate details. In contrast, the expected number of evaluations for a physical query plan plays an important role when determining the I/O and CPU costs, so we have tried to be as accurate as possible. Note that in Tables 3.5, 3.6, and 3.7, the *Tindex* operator has very simple cost formulas. Lore does not maintain fine-grained statistics about words appearing in string values, so it is impossible to predict the results of the boolean expressions supported by the *Tindex* operator. Our formulas for this operator essentially return constants.

The following definitions are used in the tables. Here p is a path expression, x is a variable, and l is a label.

- $|x|$: the number of objects expected to be bound to x. When x is bound by a path expression component then $|x| = |PathOf(x)|$ where *PathOf* takes a variable and returns the complete path expression that ends in that variable. In all other situations $|x|$ is the predicted number of evaluations as specified by Table 3.7, based on the subplan that binds x.

- $Fout_{x,l}$: estimated l-labeled fan-out for objects bound to x. Formula as given earlier in this section.

- $Fin_{x,l}$: estimated l-labeled fan-in for objects bound to x. Formula as given earlier in this section.

- $|plan|$: the expected number of evaluations returned by the subplan $plan$.

- *Selectivity*(l, *op*, *Value*): the estimated selectivity of the predicate l *op Value*, where l is a label of an incoming edge to atomic objects that must satisfy the *op Value* condition. First we predict the number of objects with incoming edge l that will satisfy the predicate using the formulas in [SAC+79]. However, since coercion may occur at run-time, we also try to predict the number of objects that will satisfy the predicate after coercion is performed. When the *value* is a string we attempt to coerce

Operator	IO Cost
Project	IOCost(Child)
Select	IOCost(Child)
NLJ	IOCost(Left) + |Left|∗IOCost(Right)
HashJoin	IOCost(Left) + IOCost(Right) + |Left| + |Right|
SMJ	IOCost(Left) + IOCost(Right)
Sort	2 * *NumRuns*(PrimVar, SecVar)
Compound	IOCost(Left) + IOCost(Right)
Scan	$Fout_{PathOf(x),l}$
Lindex	$2 + Fin_{PathOf(y),l}$
Pindex	*LengthOf*(PathExpression) + |PathExpression|
Bindex	$\|l\| * 2$
Vindex	$BLevel_{label,type1} + Selectivity_1(label, Op, Value)$ $+ BLevel_{label,type2} + Selectivity_2(label, Op, Value)$
Tindex	2
Name	IOCost(Child) + |Child| ∗ 2
Once	IOCost(Child)
CreateTemp	IOCost(Child) + |Child|∗2∗(1 + |SecVar|)
Set	IOCost(Left) + IOCost(Right) + *Struct*(Left) + *Struct*(Right)
Deconstruct	|SourceVar|∗ $(\sum_{x \in SecVar} \|x\| \ / \ \|SourceVar\|)$
ForEach	0
Aggregation	IOCost(Child) + 1
Update	IOCost(Left) + (|Left| * (IOCost(Right) + 1))
Oem	IOCost(Child) + |Child|
With	IOCost(Child) + 1
Arith	IOCost(Left) + IOCost(Right)

Table 3.5: I/O cost formulas for physical query plan nodes

Operator	CPU Cost
Project	CPUCost(Child) + (|Child| $*$ CPUCompCost)
Select	CPUCost(Child) + |Child| $*$ *EvalCost*(Predicate)
NLJ	CPUCost(Left) + |Left| $*$ CPUCost(Right) $*$ *EvalCost*(Predicate)
HashJoin	CPUCost(Left) + CPUCost(Right) + 2$*$ (*Bucketize*(Left) + *Bucketize*(Right))
SMJ	CPUCost(Left) + |Left| + CPUCost(Right) + |Right|
Sort	2$*$ (*NumRuns*(PrimVar,SecVar) + |PrimVar| + (|SecVar| / |PrimVar|))
Compound	CPUCost(Left) + CPUCost(Right) + (*CompoundResults*(Left, Right, Op) $*$ CPUCompCost)
Scan	$\|x\|$ $*$ CPUCompCost
Lindex	CPUHashCost + $Fin_{PathOf(x),l}$$*$ CPUCompCost
Pindex	$LengthOf(p)$
Bindex	$\|l\|$ $*$ CPUCompCost
Vindex	IOCost $*$ CPUCompCost
Tindex	CPUTindexLookupCost
Name	CPUCost(Child) + |Child| $*$ CPUCompCost
Once	CPUCost(Child) + (|Child| $*$ CPUHashCost)
CreateTemp	CPUCost(Child) + (|Child| $*$ CPUHashCost) + $\|SecVar\|$
Set	CPUCost(Left) + CPUCost(Right) + *Bucketize*(Left) + *Bucketize*(Right)
Deconstruct	*NumEvals*(SourceVar) $*$ ($\sum_{x \in SecVar} \|x\|$) / *NumEvals*(SourceVar)
ForEach	($\sum_{x \in SecVar} \|x\|$) / *NumEvals*(PrimVar)
Aggregation	CPUCost(Child) + (|Child| $*$ CPUAggrOpCost)
Update	CPUCost(Left) + (|Left| * (CPUCost(Right) + 1))
Oem	CPUCost(Child) + |Child|
With	CPUCost(Child) + |Child|
Arith	CPUCost(Left) + CPUCost(Right) + |Left| $*$ |Right|

Table 3.6: CPU cost formulas for physical query plan nodes

Operator	Predicted Number of Evaluations
Project	|Child|
Select	|Child| $*$ *Selectivity*(Predicate)
NLJ	|Left| $*$ |Right| $*$ *JoinSelectivity*(Left,Right)
HashJoin	|Left| $*$ |Right| $*$ *JoinSelectivity*(Left,Right)
SMJ	|Left| $*$ |Right| $*$ *JoinSelectivity*(Left,Right)
Sort	|Child|
Compound	*CompoundResults*(Left, Right, Op)
Scan	$Fout_{PathOf(x),l}$
Lindex	$Fin_{PathOf(x),l}$
Pindex	$\|p\|$
Bindex	$\|l\|$
Vindex	*Selectivity*(label, op, Value)
Tindex	1
Name	|Child| $*$ *RootSelectivity*(SourceVar,Name)
Once	|Child| $*$ *SameParents*(Variable)
CreateTemp	1
Set	1
Deconstruct	*NumEvals*(SourceVar)
ForEach	$(\sum_{x \in SecVar} \|x\|\ )$ / *NumEvals*(PrimVar)
Aggregation	1
Update	Not Applicable since an update does not return a query result.
Oem	|Child|
With	|Child|
Arith	|Left| $*$ |Right|

Table 3.7: Predicted number of evaluations for physical query plan nodes

the string into a real. If the coercion is successful then we use the statistics about real values to predict additional objects that will satisfy the predicate. When the *value* is a real or integer then we also consult statistical information about strings coerced to reals (recall Chapter 2, Section 2.5.2).

- *Struct*(*plan*): the estimated number of object I/Os associated with writing any intermediate structure returned by subplan *plan*. Determined by $|plan.PrimVar|+$ $(|plan.PrimVar|*(\sum_{x \in plan.SecVar} |x|))$.

- *LengthOf*(*p*): the number of path expression components that comprise *p*.

- *P*: the page size.

- *JoinSelectivity*(*Left*, *Right*): when a join occurs between two subpaths and a single "continuous path expression" results, then *JoinSelectivity* uses statistical information about paths through the database to predict the number of paths that will satisfy the join. A continuous path expression is one where every destination variable of every component in the path expression (except the last one) is used to feed exactly one other path expression component. Otherwise, standard relational join selectivity formulas (e.g., [SAC+79]) are used as the basis for our *JoinSelectivity* function. Like the *selectivity* function, additional terms are used for the coercion cases.

- *RootSelectivity*(*SourceVar*, *Name*): computes the percentage of objects bound to *SourceVar* that are the named object *Name*. It enables us to determine how many objects will be reached in a bottom-up traversal of a path that are not the named object we are seeking. *RootSelectivity* is determined by |Root.*Name*.*PathOf*(*SourceVar*)|/ |*SourceVar*|.

- *SameParents*(*Variable*): returns the percentage of those objects that will be bound to *Variable* that are unique. In bottom-up evaluation it is necessary to predict the number of evaluations that will be passed upwards through an existentially quantified variable. *SameParents* is computed by $|Variable|^d/|Variable|$.

- *Bucketize*(*Plan*): returns the estimated CPU time to build a hash table over the temporary result returned by subplan *Plan*. Lore uses a dynamic hashing scheme and we assume (for simplicity) that there is perfect utilization of space. Determined by *Struct*(*Plan*) * CPUHashCost.

- *CompoundResults*(*Left*, *Right*, *Op*): returns the estimated number of satisfying evaluations for a given compound operator (either `and` or `or`) and two subplans. If the operator is `and` then *CompoundResults* returns the minimum of $|Left|$ and $|Right|$, otherwise when the operator is `or` it returns $|Left| + |Right|$.

- *NumEvals*(*Variable*): the *CreateTemp* operator passes up to its parent a single evaluation stored in *Variable*. This single evaluation is expanded into many evaluations when it is operated upon by the *Deconstruct* operator. *NumEvals* estimates the number of evaluations resulting from a *Deconstruct* operation by returning the estimated number of distinct bindings to the *PrimVar* for the EES stored in *Variable*.

- *NumRuns*(*PrimVar*, *SecVar*): returns the estimated number of runs required in a multi-pass sort. *PrimVar* and *SecVar* are used to estimate the size of each entry. Using these estimates, the memory allocated for the sort, and standard formulas given in [Gra93], we can estimate the number of required runs.

- *EvalCost*(*Predicate*): returns the CPU cost associated with evaluating the simple predicate. A simple predicate has the form V_1 *op* V_2 where each V_i is either a constant or a location in memory.

In addition to the above definitions, the CPU constants we use are:

- *CPUHashCost*: the CPU cost associated with hashing an in-memory value.

- *CPUCompCost*: the CPU cost associated with comparing two in-memory values.

- *CPUAggrOpCost*: the CPU cost for incrementally maintaining an aggregate value.

- *CPUTindexLookupCost*: the CPU cost associated with doing a probe in the text indexing structure for a boolean expression.

As an example of how the I/O cost formulas were derived consider the I/O formula for the *Vindex*(l, *Op*, *Value*, x) operator: $BLevel_{l,type1}+$ $Selectivity_1(l, Op, Value)$ $+BLevel_{l,type2}+$ $Selectivity_2(l, Op, Value)$. Here *BLevel* gives the height of the relevant B+-tree index, and the *Selectivity* functions are the formulas to estimate the number of satisfying results given Lore's coercion system. (Because of type coercion, multiple B+-trees need to be accessed during a *Vindex* operation.) As a second example, the I/O cost for the *Lindex*(x, l, y)

operator is $2 + Fin_{PathOf(y),l}$, where *Fin* is the fan-in statistic as defined earlier. The *Lindex* is implemented using *extendible hashing* [FNPS79], and our cost estimate assumes no overflow buckets. Thus, it requires two page fetches (one for the directory and one for the hash bucket) and one additional page fetch for every possible parent.

As examples of CPU cost and expected number of evaluations, let us consider the formulas for the *Select* and *Once* operators given in Tables 3.6 and 3.7. At run-time, the *Select* operator iteratively requests the next evaluation from its child and checks the predicate. If the predicate is *true* then the evaluation is passed up to the *Select*'s parent. Thus, the total CPU cost reflects the time to evaluate the predicate over each evaluation, and the expected number of evaluations depends upon the expected number of evaluations from the child and the selectivity factor of the predicate. The *Once* operator only passes up an evaluation received from its child when the object bound to *Once*'s variable has not been bound before. Lore uses a temporary in-memory hash table to track the objects seen so far, so the CPU cost is reflected by the $|\text{Child}| *$ CPUHashCost term. The expected number of evaluations is determined by the number of evaluations returned by the child plan along with the expected number of duplicates that will be seen, determined by the *SameParents* function described previously.

3.4.4 Plan Enumeration

The search space of physical query plans for a single Lorel query is very large. For example, a single path expression of length n can be viewed as an n-way join, where as "join methods" Lore considers three different standard relational joins. In addition, the path index can be used to evaluate all or some of a path expression. Furthermore, there may be many interrelated path expressions in a single query, along with other constructs such as set operations, subqueries, aggregation, etc. In order to reduce the search space as well as the complexity of our plan enumerator, we use a greedy approach to generating physical query plans. Each logical query plan node makes a locally optimal decision, creating the best physical subplan for the logical plan rooted at that node. The decision is based on a given set of bound variables passed from the node's parent. The key to considering a variety of different physical plans is that a node may ask its child(ren) for the optimal physical query subplan many times, using different sets of bound variables each time. While this greedy approach greatly reduces the search space, it still explores an exponentially large number of physical query plans. Thus, our plan enumerator uses the following additional heuristics to further prune the search space.

- The optimizer does not consider joining two path expression components together unless they share a common variable. This restriction substantially reduces the number of ways to order the evaluation of path expression components.

- The *Pindex* operator is considered only when a path expression begins with a name, and no variable except the last is used elsewhere within the query. The latter requirement is based on the fact that *Pindex* only binds the last variable in its path expression, so other needed variables in the path would have to be discovered by some additional method.

- If a variable is encapsulated in a temporary result by a *CreateTemp* operator and we subsequently need to access its bound objects, we always deconstruct the temporary result and never consider plans that rediscover the original variable bindings or project out the variable during the *CreateTemp*.

- The *select* clause always executes last, since in nearly all cases it depends on one or more variables bound in the *from* clause. Also, the physical query plan will always execute either the complete *from* or complete *where* clause before moving on to the other one.

- The optimizer does not attempt to reorder multiple independent path expressions.

We now discuss how physical plans are produced. As mentioned earlier, each logical plan node creates an optimal physical plan given a set of bound variables. During plan enumeration we track for every variable in the query: (1) whether the variable is bound or not; (2) which plan operator has bound the variable; (3) all other plan operators that use the variable; (4) whether the variable is stored within a temporary result. For instance, the logical query plan node *Discover* for the path expression component "`m.Age a`" may be asked to create its optimal plan given that m has already been bound by some other physical operator, in which case it may decide that *Scan* is optimal. However, if a was bound instead then it may decide that *Lindex* is optimal. After a node creates its optimal subplan, the new state of the variables and the optimal subplan are passed to the parent. Note that a logical node may be unable to create any physical plan for a given state of the variables if it always requires some variables to be bound. In this case, "no plan" is returned and a different choice must be made at a higher level in the plan.

We provide a brief description of how each type of logical plan node generates its optimal physical subplan. Recall that the procedure to build a physical plan is recursive: a logical query plan node will ask its child(ren) for its optimal physical plan in order to construct its own optimal plan.

- *Project*, *CreateTemp*, *Set*, *Compound*, *Arith*, *With*, and *Aggr*: simply returns the corresponding physical operator over the optimal physical query plan(s) for its child(ren).
- *Select*: If the variables appearing in the selection condition are all bound then returns the *Select* physical operator over the optimal plan for the child. If the variables are not bound, then returns the *Vindex* physical operator with no subplan if the appropriate indexes exist. Otherwise returns "no plan".
- *Glue*: Creates the optimal physical query plan corresponding to the left-then-right child execution order and compares it with the optimal physical plan for the right-then-left child execution order. The cheaper plan is returned.
- *Name*: Returns a *Scan* physical operator if the *variable* has not been bound, otherwise returns a *Name* physical operator.
- *Chain*: Creates the optimal physical query plan corresponding to the left-then-right child execution order and a second optimal physical query plan corresponding to the right-then-left child execution order. A *Pindex* physical operator for the entire path expression is created if the heuristics allow it and the path index exists. The *Chain* node returns the cheapest of the considered physical query plans.
- *Discover*: If the *SourceVar* is bound then a *Scan* physical operator is created, otherwise a *Lindex* operator is created if the index exists. This operator is then compared against the *Bindex* physical operator (if the index exists). The cheapest considered is returned.
- *Exists*: Returns an *Aggregation* subplan[1] over the optimal physical plan for the child if the *Variable* is bound via a *Scan* physical operator, otherwise returns an *Exists* physical operator.

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

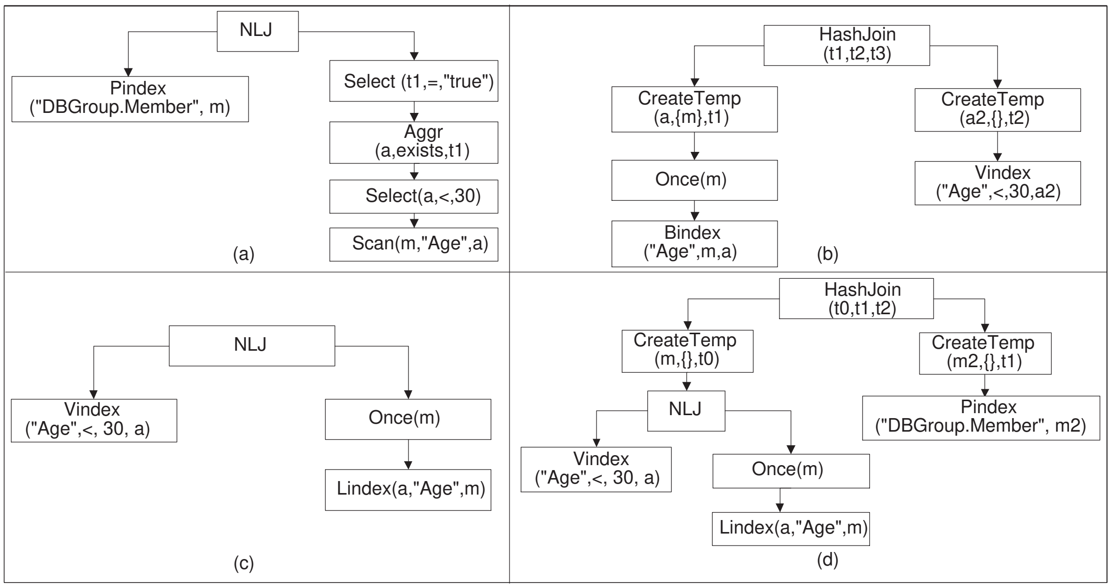

Figure 3.8: Possible transformations for Query 3.3.1 into a physical query plan

To illustrate the transformation from a logical plan to a physical plan, let us consider part of the search space explored during the creation of the physical query plan for Query 3.3.1, whose logical query plan was given in Figure 3.4. The topmost *Glue* node (indicating a rotation point) in Figure 3.4 is responsible for deciding the execution order of its children: either left-then-right or right-then-left. It requests the best physical query plan from the left child and then, using the returned bindings, requests the best physical query plan from the right child. One possible outcome is the physical query plan fragment shown in Figure 3.8(a). After exploring left-then-right execution order, the topmost *Glue* node considers the right-then-left order. The right child is another *Glue* node which recursively follows the same procedure. Suppose that for this nested *Glue* node, the left-then-right execution order results in the physical subplan shown in Figure 3.8(b), while the right-then-left execution order results in Figure 3.8(c). Suppose plan (c) is chosen based on a lower estimated cost. The bindings provided by this subplan are then supplied to the left child of the topmost *Glue* node to create the optimal query plan for the left child, which could result in the final subplan shown in Figure 3.8(d). Notice that in the right subplan for the topmost *Glue* node, the *Chain* node decided that the *Pindex* operator is the best way to get all "`DBGroup.Member m`" objects within the database, despite the fact that we already have a binding for m. This choice makes sense when the estimated fan-in for m

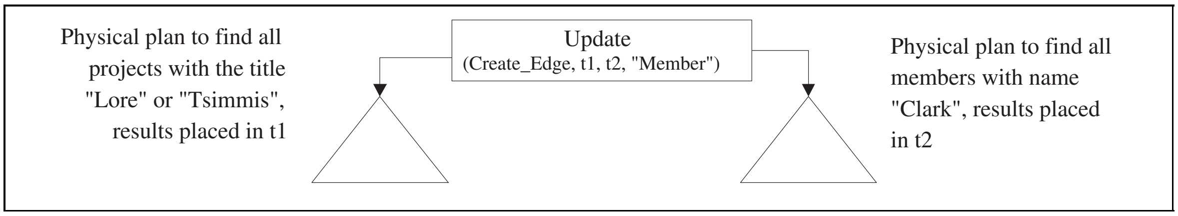

Figure 3.9: Update query plan

with label DBGroup is very high. As a final step the topmost *Glue* node decides which query plan is cheaper, either (a) or (d), and passes that plan to its parent.

3.4.5 Update Query Plans

The optimization and execution of an update statement is accomplished largely by using existing components discussed in the previous sections. We illustrate the overall approach to executing an update statement using the following example, intended to execute over the Database Group database introduced in Chapter 2 (Section 2.3) and shown in Figure 2.2. This update adds the database group member with name "Clark" as a member to both the Lore and Tsimmis projects.

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

The general form of the physical plan for this update statement is shown in Figure 3.9. An *Update* physical operator always appears as the root of a physical query plan for an update statement. In Figure 3.9, the left subtree finds those projects with title "Lore" or "Tsimmis" and the right subtree finds all members with the name "Clark". The evaluations returned by the left and right subtrees are used by the *Update* node to perform the actual update operation; valid operations are *Create_Edge*, *Destroy_Edge*, *Replace_Edge*, and *Modify_Atomic*. In our example, the *Update* node creates an edge labeled *Member* between each pair of objects identified by its subtrees. Clearly a number of improvements are possible in update processing. For instance, in our example the right subtree of the *Update* node is uncorrelated with the left subtree and thus needs to be executed only once. We currently perform this particular optimization.

3.5 Experimental Results

The query optimization techniques described in this chapter are fully implemented and integrated into Lore, including the physical operators, statistics, cost formulas, logical query plan generation, and physical query plan enumeration and selection. The implementation for the components described in this chapter consists of approximately 40,000 lines of C++ code. We also have implemented mechanisms for instructing the optimizer to favor certain types of plans (in order to debug and conduct experiments), and we have a very useful query plan visualization tool. We now present some preliminary performance results showing that our cost model is reasonably accurate and that the optimizer is choosing good plans.

All of the experiments in this chapter were run on a Sun Ultra 2 with 256 megabytes of RAM. However, Lore was configured to have a small buffer size of approximately 150K bytes, in order to match the relatively small databases used by our performance experiments. Each query was run with an initially empty buffer. Over all of the queries in our experiments the average optimization time was approximately 1/2 second.

For the experiments we used the Movie database described in Chapter 2 (Section 2.3) and shown in Figure 2.3. Recall that the Movie database contains information about movies, actors and actresses, plot summaries, directors, editors, writers, etc. The database loaded into Lore is about 5 megabytes. Vindex, Lindex, and Bindex indexes (recall Chapter 2, Section 2.5.2) account for an additional 8.1 megabytes.

Lore allows us to turn off all pruning heuristics temporarily, in order to create and execute all possible query execution plans within our search space for a single query. Thus, we can evaluate how the chosen plan performs against other possible plans. However, it is infeasible to perform this extensive experiment for large queries, since the number of plans in the search space is very large, and some plans are extremely slow to execute (even if the chosen plan is very efficient).

Experiment 3.5.1 We begin with the simple query: `Select Movies.Movie.Title`. Using exhaustive search Lore produces eleven different query plans, with estimated I/O costs and actual execution times (in seconds) as shown in Table 3.8.
In this and subsequent tables the plan chosen by the optimizer when the pruning heuristics are used is marked with a star (*). The first and best plan simply uses Lore's path index to quickly locate all the movie titles. The second plan, which is only slightly slower, uses top-down pointer-chasing. The worst plan uses *Bindex* operators and hash joins.

Plan #	Execution time (sec.)	Estimated I/O
1*	0.36	1975
2	1.78	3944
3	2.02	3944
4	14.44	9853
5	61.82	31918
6	67.24	31918
7	74.09	11823
8	94.15	37827
9	250.61	17742
10	397.18	17733
11	423.34	23855

Table 3.8: Results for Experiment 3.5.1

To evaluate the relative accuracy of our cost model, consider the estimated I/O cost against the actual execution time. With some exceptions, the estimated cost accurately reflects the relative execution time for each plan. Since our cost model is still quite simplistic, we are very encouraged by these results. □

Experiment 3.5.2 This experiment used the following query to retrieve all comedic films:

```
select Movies.Movie
where Movies.Movie.Genre = "Comedy"
```

This query is a "point" query, since many movies don't have a `Genre` subobject and most aren't comedies. Estimated I/O costs again reflected relative execution times fairly accurately, so hereafter we focus only on execution times. Twenty-four plans were considered using exhaustive search. Table 3.9 describes some of them, where "plan rank" indicates rank by execution time among all plans considered. Since the *where* clause is very selective, the optimal plan uses a bottom-up strategy: a *Vindex* operator locates all objects having the value "Comedy" and an incoming edge labeled `Genre`. The *Lindex* operator matches the remainder of the path expression in reverse. The second-best plan is only slightly slower. It uses the *Bindex* to locate all `Genre` edges, filters using the "Comedy" predicate, then proceeds bottom-up. The slowest plan uses a poor combination of *Bindex* operators and joins. Top-down evaluation results in the seventh-best plan. □

Plan Rank	Execution Time	Description
1*	0.3307	Bottom-up
2	0.3768	*Bindex* for `Genre` with *Select* then *Lindex*
7	3.3384	Top-down
24	458.58	*Bindex* with hash joins

Table 3.9: Results for Experiment 3.5.2

Plan Rank	Execution Time	Description
1	0.33	Bottom-up
2*	3.68	Top-down
3	6.95	Hybrid with *Pindex*
4	7.01	Hybrid with pointer-chasing
5	23.13	*Bindex* over Movie with *Vindex* then *Lindex*

Table 3.10: Results for Experiment 3.5.3

Experiment 3.5.3 In the remaining two experiments we did not execute all possible plans since the queries and space of plans are much larger. Instead, we generated and executed a sampling of plans from within our search space. Again, the plan chosen by the optimizer is marked with a star (*). For this experiment we executed the following query, which retrieves all names of actors who had the lead role in a film. Note that the query contains two existentially quantified variables.

```
select n
from Movies.Actor a, a.Name n
where exists f in a.Film: exists r in f.Role: r = "Lead"
```

Results are shown in Table 3.10. Notice that the optimizer chose plan 2, the top-down or pointer-chasing execution strategy, as the best plan. The mistake is due largely to simplistic estimates of atomic value distributions (histograms or more detailed atomic value statistics would help) and of set operation costs. □

Experiment 3.5.4 In this experiment we issued the following query, shown below, which selects movies with a high quality rating.

Plan Rank	Execution Time	Description
1*	0.61	*Bindex* for `Rating`, then *Lindex* up
2	0.89	Bottom-up
3	4.04	Top-down

Table 3.11: Results for Experiment 3.5.4

```
select Movies.Movie
where Movies.Movie.Rating > 8;
```

We considered only a sampling of all possible plans, again due to the large plan space size. Results are shown in Table 3.11. Since it turns out that quality ratings are fairly uncommon in the database, the optimizer (correctly) chooses to find all ratings via the *Bindex*, then to work bottom-up. □

In general, the experiments reported here (along with others conducted) allow us to conclude: (i) our cost estimates are accurate enough to select the best plan in many situations; (ii) execution times of the best and worst plans for a given query and database can differ by many orders of magnitude; and (iii) the best execution strategy is highly dependent on the query and database, indicating that a cost-based query optimizer for semistructured data is crucial to achieving good performance.

3.6 Related Work

Other semistructured databases. The *UnQL* query language [BDHS96, FS98], introduced in Chapter 2 (Section 2.6), performs query optimization by defining a translation from UnQL to *UnCAL* as described in [BDHS96]. This translation provides a formal basis for deriving optimization rewrite rules such as pushing selections down. However, UnQL does not have a cost-based optimizer as far as we know. *Strudel*, introduced in Chapter 2 (Section 2.6), considers query optimization in [FLS98]. In Strudel, semistructured data graphs are introduced for modeling and querying, while the data itself may reside elsewhere in arbitrary format. A key feature of Strudel's approach to query optimization is the use of declarative *storage descriptors*, which describe the underlying data stores. The optimizer enumerates query execution plans, with a cost model that derives the costs of operators from their descriptors. In contrast, Lore data is stored under our control, and the user may dynamically create indexes to provide efficient access methods depending upon the expected queries. Finally, [FLS98] includes detailed experimental results of how large their search space is, but no other performance data is given. In contrast, our experiments focus on the performance of the query plan selected by the optimizer versus other possible query plans.

Earlier systems, such as *Multos* [BRG88] and *Model 204* [O'N87], both introduced in Chapter 2 (Section 2.6), considered query optimization over data similar to semistructured data. Multos operated on *complex data objects* which allowed, among other things, sets and pointers to objects of any type. Basic knowledge of the schema was crucial to query compilation, however, and queries were placed into categories with a fixed set of execution strategies for each category. Lore follows a more traditional and flexible model of query processing. Model 204 was based on self-describing record structures somewhat resembling OEM, but the work concentrated primarily on clever bit-mapped indexing structures to achieve high performance for its relatively simple queries.

Relational databases. As discussed in Section 3.1, at a coarse level the problem of optimizing a Lorel path expression is similar to the join ordering problem in relational databases. However, join ordering algorithms usually rely on statistics about each joining pair, while for typical queries in our environment it is crucial to maintain more comprehensive statistics about entire paths. The computation and storage of our statistics is further complicated by the lack of a schema. In addition, when quantification is present in our queries, the SQL translation results in complex subqueries that many relational optimization frameworks are ill-equipped to handle.

Object-oriented databases. There has been some work on optimizing path expressions in an OODBMS context. [GGT96] proposes a set of algorithms to search for objects satisfying path expressions containing predicates, and analyzes their relative performance. Our work differs in that we consider many interrelated path expressions within the context of an arbitrary query with other language constructs. We also provide additional access methods for path expressions, and our optimization techniques are implemented within a complete DBMS. Similar comparisons can be drawn between our work and other recent OODB optimization work, e.g., [GGMR97, KMP93, OMS95, SO95]. Some recent papers have specified cost models for object-oriented DBMSs, e.g., [BF97, GGT95]. Object-oriented databases typically support object clustering and physical extents, rendering many of these formulas inapplicable in our setting.

General path expressions. Other recent work, including [FLS98, CCM96], has considered the problem of optimizing the evaluation of *generalized path expressions* (similar to our general path expressions). In [CCM96] an algebraic framework for the optimization of general path expressions in an OODBMS is proposed, including an approach that avoids exponential blow-up in the query optimizer while still offering flexibility in the ordering of operations. In [FS98] two optimization techniques for general path expressions are presented, *query pruning* and *query rewriting using state extents*. In this chapter we described the default way in which Lore evaluates general path expressions. The work of [CCM96, FS98] could be applied within our query optimization framework, since both papers describe query rewrites that could be incorporated as transformations over Lore's logical query plan.

Chapter 4

Query Rewrite Transformations

Recall from Chapter 3 that *query rewrites* are transformations from a logical query plan to a different but semantically equivalent logical query plan. The hope is that the rewritten logical query plan will result in a more efficient physical query plan than the original. In this chapter we present two query rewrites that are not currently found in relational or object-oriented DBMSs. Both rewrites are geared towards improving the evaluation of non-trivial path expressions. The first rewrite technique removes regular expression operators from a general path expression. The second rewrite technique pushes portions of the *where* clause into the *from* clause.

The first query rewrite technique presented in this chapter appeared in [MW99a].

4.1 Introduction and Motivation

A query rewrite transforms a query into an equivalent query that may be more amenable to optimization by later stages of the query compiler. A query rewrite could be performed over the textual representation of the query, or over the parse tree, but it is often most convenient to transform the logical query plan. In this chapter we introduce two separate and complementary query rewrite techniques that are unique to semistructured data. Neither of these rewrites are performed in relational or object-oriented DBMSs, although the second rewrite could potentially be incorporated into an OODBMS.

The first rewrite, presented in Section 4.2, uses the DataGuide (recall Chapter 2, Section 2.5.3) to remove regular expression operators from a general path expression. In Chapter 3 (Section 3.4.2) we discussed Lore's default manner of evaluating path expressions containing regular expression operators. Recall that the default evaluation strategy may needlessly explore large amounts of data. By rewriting a general path expression we may prune portions of the database from run-time consideration at the cost of time spent performing the rewrite. The decision of when this rewrite is advantageous is influenced by the size of the data in relation to the size of the DataGuide, and where the regular expression operators appear in the context of the query.

The second rewrite, presented in Section 4.3, optimizes *graph-structured* path expressions. A graph-structured path expression is a branching path expression together with an oid equality condition in the *where* clause. Recall from Chapter 2 (Section 2.4.7) that a branching path expression explores two or more separate paths through the data. In the basic Lore optimizer as described in Chapter 3, graph-structured path expressions are evaluated by first binding the separate paths and then performing the oid comparison. We introduce in Section 4.3 a query rewrite that enables variable bindings from one path to be "passed" to another path, in many cases eliminating the full cross-product between the path bindings.

We discuss related work for both query rewrites in Section 4.4.

4.2 Rewriting General Path Expressions

General path expressions, introduced in Chapter 2 (Section 2.4.3), use regular expression operators to specify a *path pattern.* Recall that general path expressions are particularly useful when the structure of the data is irregular, changes often, or is not completely known to the user. Run-time evaluation of general path expressions can be very expensive, since they may involve exploration of significant portions of the database (recall Chapter 3, Section 3.4.2). In this section we discuss improving efficiency by performing *compile-time expansion* of general path expressions based on the DataGuide of the current database. Compile-time expansion incurs the cost of exploring the DataGuide and rewriting the query, but it can eliminate significant amounts of unnecessary database exploration at run-time. We have implemented our algorithms in Lore and performance results confirming the benefits of our approach are reported.

As an example of a query containing a general path expression, consider the following query intended for the the Library Database introduced in Chapter 2 (Section 2.3, Figure 2.5).

Query 4.2.1

```
select a
from Library.# x, x.Author a
where x.Title like "%stand%"
```

Recall from Chapter 2 (Section 2.4.3) that # is an abbreviation for `(.%)*`, which matches zero or more edges with any label. Query 4.2.1 uses the SQL-style "like" operator to return all authors of objects located somewhere in the subgraph rooted at `Library` that have "stand" in their title. We will use this query in the examples for the remainder of this section.

We now introduce two schemes for eliminating general path expressions by query rewriting at compile-time: *path expansion* (for eliminating *, +, and ? operators) and *alternation elimination* (for eliminating | operators).

4.2.1 Path Expansion

We use the DataGuide to help us replace all regular expression operators *, +, ? with alternations (|) representing possible label paths in the current database. The rewrite algorithm effectively executes the entire general path expression over the DataGuide, recording the label paths that are seen. We then replace the general path expression in the query with the set of all possible matching label paths. Performing this replacement can eliminate unnecessary exploration of the database at run-time for most of the query execution plans that might be selected by Lore's optimizer. Furthermore, path expansion cannot increase query execution time for any plan, including plans that use Lore's path index (recall Chapter 2, Section 2.5.2). Cycles must be handled carefully: the semantics of Lorel are to traverse a cycle in the database no more than once when evaluating a path expression with a closure operator. The same semantics must be preserved when we eliminate closure at compile-time.

Lore uses a "compile-and-run" scenario. Concurrency control on the DataGuide can ensure that the structure of the database remains stable between compilation and execution. If the DataGuide can change between compile-time and run-time then our approach does not work.

Example 4.2.1 Consider the subpath "`Library.# x, x.Author a`" in Query 4.2.1. As part of preprocessing (recall Chapter 2, Section 2.5.1) Lore translates # into (.%)* and then into $(.l_1|...|.l_n)^*$ for all labels l_i in the database. Thus, the subpath $\langle Library,\#,x\rangle$ binds to x any descendant of the named object `Library`. This subpath is further restricted by the second path expression component $\langle x$, Author, $a\rangle$. By performing compile-time expansion of # (or equivalently $(.l_1|...|.l_n)^*$) we can ignore at run-time those database paths that don't lead to an `Author` subobject. For example, by applying this path to the DataGuide as shown in Figure 2.5, we determine that the # can match only `Proceedings.Conference.Paper`, `Books.Book`, and `Movies.Movie.BasedOn`, yielding the equivalent path expression "`Library(.Proceedings.Conference.Paper|.Books.Book|.Movies.Movie.BasedOn) x, x.Author a`." □

By expanding a general path expression at compile-time using the DataGuide, we are guaranteed to visit, at run-time, a subset of the objects we would have visited with the original path expression, regardless of query execution strategy. If the DataGuide is small and resides in memory, then the expansion itself will be very fast and almost certainly worthwhile. However, when the DataGuide is large and may reside partially (or completely) on disk, it is less obvious that the cost associated with compile-time expansion, plus the cost of evaluating the expanded path expression, will be less than the cost of run-time evaluation of the original path expression as described in Chapter 3 (Section 3.4.2). In Section 4.2.3 we evaluate empirically the expansion tradeoffs. Developing an algorithm to decide efficiently when to perform path expansion is an area of future work.

In the special case where path expansion results in no matched paths in the DataGuide, we effectively "cancel" path expansion and leave the path expression in its original form. In some situations we could use the information that there are no matching paths to avoid executing the query entirely, and in other situations we could avoid the execution of a disjunction in the *where* clause or the execution of a subquery. However, the exact effect for a given query is quite complex and requires a case-by-case handling of all places a path expression can appear in a query. We do not consider the issue further in this thesis.

Expansion of label wildcards (recall Chapter 2, Section 2.4.3) could use the same technique that was described here for the expansion of regular expression operators. Recall from Chapter 2 (Section 2.5.1) that labels with wildcards are expanded based on the list of all labels appearing anywhere in the database. We could use the DataGuide to expand a label containing wildcards into the set of matching labels that appear in the data in the context of the enclosing path expression. Similarly, the DataGuide could be used to reduce the number of alternations appearing in a subpath. For example, in the path expression "`Library(.Authors|.Author|.Penname).Address`" the DataGuide could be used to reduce the alternation to "`Library(.Authors|.Author).Address`" if no "`Library.Penname.Address`" appeared in the DataGuide. We do not consider these additional uses of path expansion further, but the next section addresses issues related to the alternation operator.

4.2.2 Alternation Elimination

We can eliminate alternation operators in general path expressions by introducing either a `union` operator or a disjunct in the `where` clause. If the alternation appears in the *from* clause, e.g., "`From Library(.Book|.Movie) x, x.Title y`", then we can rewrite this clause as "`From ((Library.Book) union (Library.Movie)) x, x.Title y`". This transformation can be applied as many times as necessary, and if `union` is implemented properly it will not introduce any computational or I/O overhead to any execution strategies. Once an alternation is replaced with a `union` we can consider the following query rewrite:

$$\begin{array}{l}\text{Select s}\\ \text{From ...,((x.Book) }\mathtt{union}\text{ (x.Movie)) y,...}\\ \text{Where w}\end{array} \rightarrow \left(\begin{array}{l}\text{Select s}\\ \text{From ...,x.Book y,...}\\ \text{Where w}\end{array}\right) \mathtt{Union} \left(\begin{array}{l}\text{Select s}\\ \text{From ...,x.Movie y,...}\\ \text{Where w}\end{array}\right)$$

Here we have replaced the `union` expression in the *from* clause with two queries connected via a `union`. In Section 4.2.3 we illustrate when this transformation is advantageous.

When the alternation appears in the *where* clause, in some cases we can rewrite it using an explicit `or` operator. For example, "`Where exists y in x(.Subject|.Keyword): y='DB'`" can be rewritten as: "`Where exists y in x.Subject: y='DB' or exists z in x.Keyword: z='DB'`. The advantage of this transformation is that it allows the optimizer to take advantage of an index created over `Keyword` objects when no corresponding index exists for `Subject` objects, or vice-versa. We can introduce disjunction in this fashion only when the query is expressed in conjunctive normal form (CNF) as defined in Chapter 2 (Section 2.4.5). The general case of the transformation relies on subtle Lorel semantics not covered in this thesis, but the general idea is as illustrated by the simple example above.

As another example, recall Query 4.2.1 which in Example 4.2.1 we rewrote as:

```
select a
from Library(.Proceedings.Conference.Paper|.Books.Book|.Movies.Movie.BasedOn) x,
   x.Author a
where x.Title like "%stand%"
```

$$\left(\begin{array}{l}\text{select a}\\ \text{from Library.Proceedings.Conference.Paper x, x.Author a}\\ \text{where x.Title like "\%stand\%"}\end{array}\right) \quad \text{union}$$

$$\left(\begin{array}{l}\text{select a}\\ \text{from Library.Books.Book x, x.Author a}\\ \text{where x.Title like "\%stand\%"}\end{array}\right) \quad \text{union}$$

$$\left(\begin{array}{l}\text{select a}\\ \text{from Library.Movies.Movie.BasedOn x, x.Author a}\\ \text{where x.Title like "\%stand\%"}\end{array}\right)$$

Figure 4.1: Alternation elimination for Query 4.2.1

We can remove the alternation operators by replacing them with `union` expressions and then break the *from* clause into three separate queries as shown in Figure 4.1.

4.2.3 Experimental Results

In this section we empirically evaluate the benefits of the compile-time rewrites described in Sections 4.2.1 and 4.2.2. We have extended the Lore system to expand path expressions containing *, +, and ?, using the DataGuide. Lore does not support a general query rewrite mechanism to implement the `union` rewrite described in Section 4.2.2, so we have hand-fed the original and rewritten queries in order to evaluate the effectiveness of the transformation.

Path Expansion

In our first set of experiments we compare execution times for path expressions containing # with and without path expansion. These experiments continue to use the Library database (Chapter 2, Section 2.3 and Figure 2.5). Recall that our database generator creates Library databases using parameters such as number of books, average number of authors, percentage of books that are sequels, etc. In Tables 4.1 and 4.2 we present execution times (in seconds) for path expression evaluation, with and without path expansion, for a variety of path expressions. In all cases, the query execution strategy used is the top-down query execution strategy (recall Chapter 3, Section 3.3). Other execution strategies might show different levels of improvement, but again our rewrite will never degrade the performance of the final query execution plan. The experimental results shown in Table 4.1 were generated by running Lore over a small version of the Library database with 31,028 objects and 42,270

Path Expression	Compile-time Expansion	Query Execution	Total	Execution of Original Path
Library.#.Name	0.35	28.74	29.09	54.10
Library.#.Title	0.21	10.0	10.21	33.43
Library.Books.Book.#.Name	0.07	5.37	5.44	10.05
Library.Movies.#.Title	0.1	2.84	2.94	12.30
Library.Movies.Movie.Actor.#	0.01	3.07	3.08	3.12

Table 4.1: Path expansion—execution times for small Library database

Path Expression	Compile-time Expansion	Query Execution	Total	Execution of Original Path
Library.#.Title	0.77	102.65	103.42	412.13
Library.Books.Book.#.Name	0.18	99.10	99.28	126.74
Library.Movies.#.Title	0.24	83.73	83.97	223.10
Library.Movies.Movie.Actor.#	0.01	35.4	35.41	37.93

Table 4.2: Path expansion—execution times for larger, cyclic Library database

edges. In this database there are 2,000 books, 4,000 authors and actors, and 1,000 movies. The data is not tree-structured, but there are few cycles in the graph, and the DataGuide consists of about 100 objects. For the experimental results given in Table 4.2 the database contains 132,727 objects and 245,335 edges, with 10,000 books, 20,000 authors and actors, and 10,000 movies. Even though the data follows the same general form as the first database, the data is generally more cyclic, and the DataGuide is about double the size of the first DataGuide.

Tables 4.1 and 4.2 show that compile-time path expansion can reduce overall execution time by up to 75%. For these databases and DataGuides, the time to perform expansion is dwarfed in all cases by actual query execution time. As can be seen, there is essentially no benefit to expanding the # operator when it appears at the end of the path (the last row in each table) since in this case expansion does not prune any paths from consideration at run-time. (The small difference in query execution times is due to a more efficient implementation of the physical *Scan* operator used by the transformed path expression.)

Clearly we can construct a database with a very large DataGuide, where the cost of exploring the DataGuide does outweigh the run-time benefit of compile-time expansion. For example, we generated a database whose DataGuide's size was close to the size of

Key	Query
A	Select x From Library.Books y, y.Book x where x.(Title|Keyword) = "Armageddon"
B	(Select x From Library.Books y, y.Book x Where x.Title = "Armageddon") `Union` (Select x From Library.Books y, y.Book x Where x.Keyword = "Armageddon")

Table 4.3: Key for Table 4.4

Query	Execution Time	Notes
A	5.9	No index used.
B	4.5	Index created and used over `Keyword`. No index over `Title`.
B	4.9	Index created and used over `Title`. No index over `Keyword`.
A	0.017	Index created and used for both `Title` and `Keyword`.
B	0.019	Index created and used for both `Title` and `Keyword`.

Table 4.4: Execution times for alternation elimination

the database due to very unstructured data. The time to execute a sample general path expression was 156 seconds, which was faster than the time required to expand the path expression (46 seconds) and then execute the expanded path expression (150 seconds).

Alternation Elimination

To test the effectiveness of replacing alternation with `union`, we ran the experiments reported in Tables 4.3 and 4.4. Table 4.3 shows our original (A) and rewritten (B) queries. We present the transformation of alternation in the *where* clause into a `union` operator (with subsequent query rewrite), rather than disjunction, because the performance improvement is more significant. Table 4.4 shows execution times (in seconds) with a variety of indexes over the smaller library database.

The experiments in Table 4.4 show one situation in which it is beneficial to eliminate alternation. The Vindex can be used to quickly locate atomic objects with specific values and incoming labels (e.g., objects with incoming label `Title` and value "Armageddon"). We can then traverse backwards through the graph to match the path expression being evaluated. In our example queries, if a value index exists for `Title` or `Keyword` objects but not both, then it would be extremely difficult for the optimizer to exploit just one index in the evaluation of Query A. Query B, however, can take advantage of a single index when it exists. For example, with an index over `Keyword` objects the rewritten query ran about 24% faster than the original. The speedup using a `Title` index was less because typically there are many `Keyword` subobjects for a movie, but usually only one `Title`, thus requiring less search when a `Title` index is not present. If both indexes are present then the optimizer selects the hybrid execution strategy (recall Chapter 3, Section 3.3) for both queries, with significantly faster execution times as shown in the last two lines of Table 4.4.

In general, the advantage of transforming alternation into a `union` expression, and then into two queries connected via a `union`, is that even though some redundant path traversals may occur in the rewritten query, its two subqueries can be optimized independently and thus can use very different execution strategies. Transforming alternation into disjunction in our particular example has a less dramatic effect. However, the same general principle of separating execution strategies applies when the path expression operands to the alternation are longer.

4.3 Meeting-Path Optimization

In this section we introduce a query rewrite technique called meeting-path optimization (MPO). The rewrite introduces a use of variables that is not valid according to the original Lorel specification. Thus, we also needed to extend the Lore system to accommodate the rewrite. MPO is very effective for a class of commonly-posed queries.

We begin by introducing motivating examples in Section 4.3.1. In Section 4.3.2 we discuss limitations of MPO. The MPO rewrite itself is presented in Section 4.3.3. We conclude in Section 4.3.4 with the presentation of some experimental results.

4.3.1 Motivating Examples

Lorel queries over graph-structured databases may contain branching path expressions that require paths in the data to meet at specific points.

Example 4.3.1 Consider the following query executed over the Movie database introduced in Chapter 2 (Section 2.3 and Figure 2.3). This query finds all pairs of female and male actors who have appeared together in a comedy.

```
select a1.Name, a2.Name
```

```
from Movies.Actor a1, a1.Film f1, f1.Movie m1,
     Movies.Actress a2, a2.Film f2, f2.Movie m2
where m1 = m2 and m2.Genre = "Comedy";
```

This query explores two paths through the Movie database. The first path finds all the movies that all actors appeared in, and the second finds all movies that all actresses appeared in. An actor and actress pair are in the result if the movies are the same and the movie is a comedy. This query is an example of a *graph-structured path expression* as described in Section 4.1. The *where* clause in this query uses oid equality `m1 = m2` to ensure that the movies bound by the two branching paths are the same object. □

Without the oid equality condition, a graph-structured path expression would require a variable to appear as the destination variable for more than one path expression component, for example "`f1.Movie m, f2.Movie m`". Lorel (as well as other query languages for semistructured data) does not allow variables to be used in this way, so in order to specify such queries, the *where* clause must contain a predicate with the interpretation `oid(x) = oid(y)`. This clause returns `true` if and only if variables x and y are bound to the exact same object. Since Lorel supports various notational and semantic shortcuts, in some cases the clause can be expressed simply as `x = y`, as illustrated in Example 4.3.1 with the expression `m1 = m2`. To execute a query of this form, Lore's original optimizer (as described in Chapter 3) was limited to generating plans that first bound $\langle x, y \rangle$ pairs before checking the predicate.

Query Execution Strategies

Let us describe and graphically depict some of the different query execution strategies possible for Example 4.3.1. We use a high-level graphical view for representing a path expression evaluation strategy which shows the structure of the branching path expression (i.e., the relationship of variables in the query) and the access methods and order of execution for the path expression components. In this view a variable in the query becomes a node in the graph and edges connect two source and destination variables of a path expression component. A dashed edge between two variable nodes indicates an oid equality comparison. A solid node indicates that a simple selection predicate is applied to a child of that variable. The order in which path expression components are executed in the plan is indicated by a circled number next to each edge. A circled number next to a solid node or a dashed line

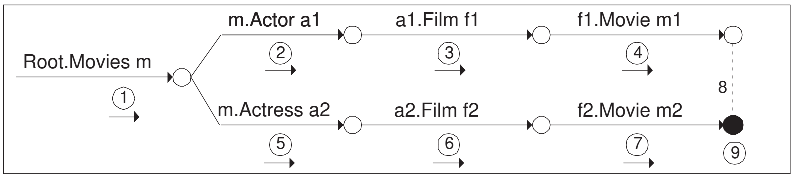

Figure 4.2: Top-down execution strategy for Example 4.3.1

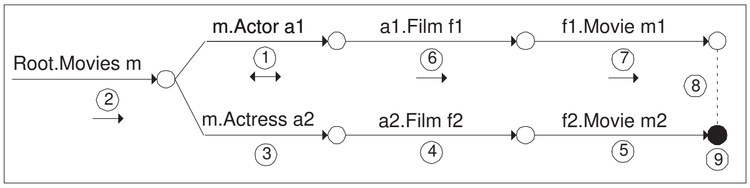

Figure 4.3: Execution strategy for Example 4.3.1 chosen by original Lore optimizer

indicates when the predicate is evaluated. A left arrow next to an order number indicates that a *Lindex* access method is used for that component, while a right arrow indicates a *Scan* access method. A double arrow indicates that a *Bindex* access method is used.

The most straightforward plan for Example 4.3.1 is a top-down plan. Recall from Chapter 3 (Section 3.3) that a top-down plan uses a depth-first search of the graph to provide bindings for all variables in the *from* clause before the *where* clause is evaluated. A top-down plan uses all *Scan* access methods, and a graphical depiction of this query plan is shown in Figure 4.2. The first step of the plan is to find the named object `Movies`. Then, in steps 2–4, the path expression components "`m.Actor a1, a1.Film f1, f1.Movie m1`" are matched using *Scan* access methods. The remaining variables in the *from* clause are similarly bound in steps 5–7. In step 8, the oid comparison is performed. Finally, step 9 checks the remaining predicate. This plan essentially computes the cross-product between all actor/movie pairs and all actress/movie pairs.

The Lore optimizer, as described in Chapter 3 and without the new optimization technique that we introduce in this section, produces a better plan for the query in Example 4.3.1 than the top-down plan, shown in Figure 4.3. This plan uses an efficient combination of *Bindex*, *Lindex*, and *Scan* access methods. However, this plan, and any other generated by the original Lore optimizer, cannot evaluate the expression `m1 = m2` until both $m1$ and $m2$ are bound by an access method.

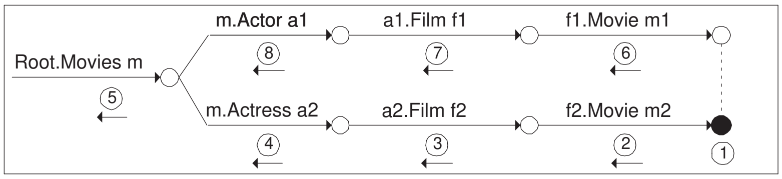

Figure 4.4: Execution strategy possible after MPO

Now consider the plan illustrated in Figure 4.4. The first step uses a *Vindex* to find all comedies in the database. In steps 2–5, a reverse evaluation of the second portion of the *from* clause is performed using *Lindex* physical operators. Steps 6–8 repeat the process for the first portion of the *from* clause. Notice that this plan does not perform a cross-product between all actor/movie and actress/movie pairs. Instead, it locates a comedy and then traverses backwards through the data looking for all actors and actresses that worked on that movie. This strategy reduces to small cross-products between actors and actresses for a single movie (which form the result of the query). In many situations this plan executes faster than both the top-down plan and the plan generated by the previous Lore optimizer. Specifically, this plan is better when the predicate `m2.Genre = "Comedy"` is selective and the amount of data explored by the reverse evaluation of the path expressions in the *from* clause is smaller than the data seen during a forward traversal of the same paths.

Note that we have not considered the use of the *Pindex* operator (recall Chapter 3, Section 3.4.2) in the context of this query rewriting technique, although it should not be difficult to incorporate.

4.3.2 Overview and Limitations

The MPO technique covers: (1) rewriting graph-structured branching path expressions to make oid equality explicit within the path expression, and (2) enabling the optimizer to take advantage of these new types of queries. Once the necessary changes are made to the optimizer then the overhead of MPO consists only of the rewrite. After the rewrite is performed, the optimizer can apply all of its techniques for optimizing path expressions to generate efficient query execution plans. The changes to the optimizer and query engine turned out to be minor, since the optimizer was already designed to handle path expressions in a very general way.

The meeting-path rewrite can only be performed on Lorel queries expressed in disjunctive normal form (DNF) as defined in Chapter 2 (Section 2.4.5). We can apply the rewrite for an $\langle x, y \rangle$ variable pair if each disjunct in the *where* clause contains either `oid(x) = oid(y)`, or `x = y` such that we know at rewrite time that all objects bound to x and y are complex objects and thus result in the `oid(x) = oid(y)` interpretation. We can determine whether all objects are guaranteed to be complex by consulting statistical information stored in the DataGuide.

4.3.3 The Meeting-Path Rewrite

The intuition behind the MPO rewrite is that we remove a predicate of the form `x = y` from the *where* clause and incorporate it into all path expression components that use x or y, allowing the generated query plans to use either variable in place of the other. MPO is somewhat related to the transitive closure of predicates in relational systems. Both use known facts about objects or values to open new optimization strategies. Our approach differs in that the transformation applied to the query does not affect only the *where* clause, but also the *from* and *select* clauses.

Once we have determined that a predicate `oid(x) = oid(y)` or `x = y` is suitable for the rewrite as specified in Section 4.3.2, the rewrite itself is very simple:

1. Remove `oid(x) = oid(y)` or `x = y` from the *where* clause.

2. Replace all occurrences of variable y with x in the remainder of the query.

The rewritten query is no longer valid Lorel since a variable is bound by more than one path expression component after the rewriting. However, the extension to the language is straightforward: when variable x appears as the destination variable in two path expression components then both path expression components must result in the same object being bound to the destination variable. The meeting-path rewrite applied to the query in Example 4.3.1 results in:

Query 4.3.1

```
select a1.Name, a2.Name
from Movies.Actor a1, a1.Film f1, f1.Movie m1
     Movies.Actress a2, a2.Film f2, f2.Movie m1
where m1.Genre = "Comedy";
```

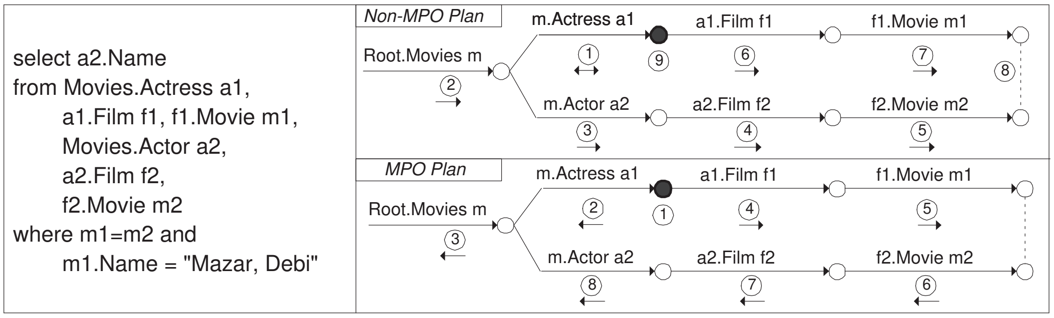

Figure 4.5: Query plans for Experiment 4.3.1

Note that $m1$ now appears as the destination variable in two path expression components.

The advantage of applying the MPO rewrite to this query is that once variable $m1$ is bound by an access method for one of the path expression components, then that binding can be used for both path expressions that end in $m1$. This allows the more efficient plan shown in Figure 4.4 to be generated.

4.3.4 Experimental Results

For our experiments we used the Movie database described in Chapter 2 (Section 2.3, Figure 2.3). The total size of the database, including all indexes, is over 10 megabytes. We include results from three queries, shown on the left sides of Figures 4.5, 4.6, and 4.7. Table 4.5 summarizes the resulting query execution times. Since MPO takes a negligible amount of time prior to query optimization (an average of 100 milliseconds), we have not incorporated that time into our results.

Query	Execution time without MPO	Execution time with MPO
1	4131.42	1.62
2	114.52	86.51
3	4386.12	17.93

Table 4.5: Experimental results for meeting-path optimization

Experiment 4.3.1 The query in Experiment 4.3.1 finds all actors that worked on a movie with Debi Mazar. The query, along with the query plans produced with and without MPO,

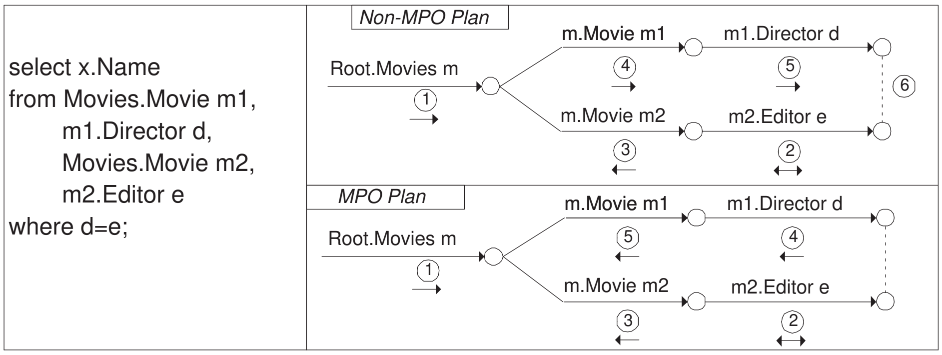

Figure 4.6: Query plans for Experiment 4.3.2

are shown in Figure 4.5. From Table 4.5 we see that MPO reduced query execution time by over three orders of magnitude. The plan produced without MPO used a combination of *Bindex*, *Lindex*, and *Scan* access methods to bind all variables in the *from* clause before checking the predicates in the *where* clause. The plan generated with MPO first uses a *Vindex* to find all `Name` objects in the database with value "Mazar, Debi" and then a *Lindex* for the `Name` (not shown in the plan), yielding a binding for $a1$. Steps 2 and 3 match the subpath "`Movies m, m.Actress a1`" using *Lindex* access methods. *Scan* operators in steps 4 and 5 match the subpath "`a1.Film f1, f1.Movie m1`". With the rewrite, $m2$ is bound by step 5 so the remaining steps of the plan use *Lindex* access methods for the remaining path expression components. □

Experiment 4.3.2 This experiment uses a query that finds all people that worked as both an actor and a director on a single movie. The query, along with the query plans produced with and without MPO, are shown in Figure 4.6. The 25% performance improvement isn't as dramatic as in the first experiment because the rewrite of the first query benefited by using the `Name` predicate early. In this experiment, MPO resulted in a slightly more efficient configuration of access methods. Figure 4.6 shows that step 1 discovers the named object `Movies`. In step 2, a *Bindex* for `Editor` is followed by a *Lindex* for `Movie`. Steps 4 and 5 use *Lindex* access methods for path expression components "`m.Movie m1, m1.Director d`". This plan contrasts with the slower plan produced without MPO, which uses a combination of *Bindex*, *Scan*, and *Lindex* access methods with a sort-merge join on the meeting point of the two paths. □

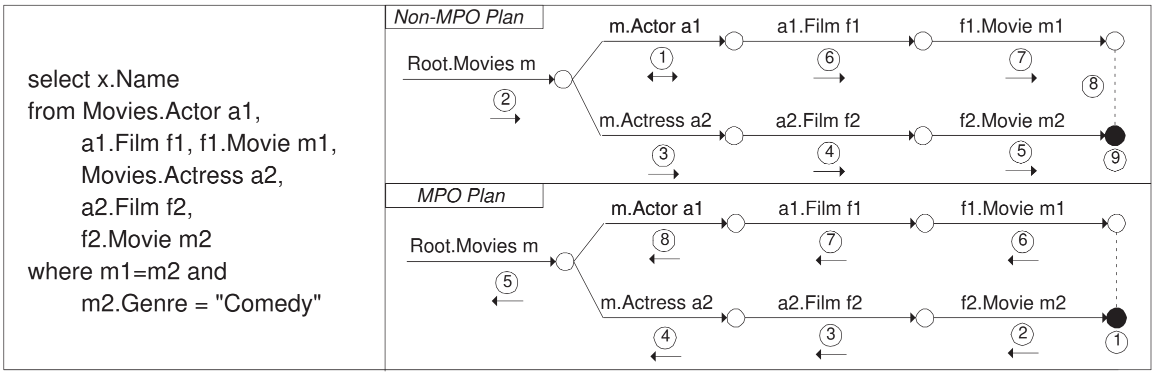

Figure 4.7: Query plans for Experiment 4.3.3

Experiment 4.3.3 The query in this experiment is the same as the query in Example 4.3.1 introduced in Section 4.3.1. This query is similar in structure to the query in Experiment 4.3.1, however the second selection condition appears at a different location in the overall graph structure of the query. The query, along with the query plans produced with and without MPO, are shown in Figure 4.7. From Table 4.5 we see that the plan with MPO runs in just under 18 seconds, while the plan without MPO takes 73 minutes. The plan generated with MPO is the efficient plan we discussed in Section 4.3.1. The plan without MPO (also discussed in Section 4.3.1) uses a combination of *Bindex*, *Lindex*, and *Scan* access methods, but must perform the cross product. □

4.4 Related Work

Our work on rewriting general path expressions is similar in spirit, but not in details, to [FS98]. In [FS98], a cross-product is computed between a *graph schema*—a summary of the database that must be small and reside in memory—and a representation of the query. From this cross-product an expanded version of the query is produced that is expected to execute more efficiently than the original. Our algorithm traverses the DataGuide (which corresponds roughly in Lore to their graph schema) in order to rewrite the query. We do not require that the DataGuide is small since a full cross-product is not formed, and we do not require that the DataGuide resides in memory. We have also introduced some query rewrites not covered in [FS98], such as the alternation elimination and the union rewrite.

The meeting-path rewrite optimizes branching path expressions with a graph shape by enabling oid equality predicates in the *where* clause to be incorporated into path expression evaluation. To the best of our knowledge ours is the first work specifically on optimizing graph-structured path expressions.

Chapter 5

Subplan Caching

Based on the query optimization framework introduced in Chapter 3, in this chapter we introduce an optimization technique called *subplan caching*. This technique introduces one or more in-memory caches to be used during execution of a physical query plan. When placed properly within a plan these caches store data that would otherwise have to be refetched from disk or recomputed many times. We introduce a generic *Cache* physical operator and extend the search space of physical query plans explored by Lore's optimizer to allow for efficient placement of the *Cache* operator. We present experimental results illustrating when the subplan caching technique is beneficial.

5.1 Background

In database query languages, a *correlated subquery* is a subquery that refers to one or more variables bound outside of the subquery. The following SQL query contains a correlated subquery, since `R.a` referenced in the subquery is bound by the outermost *from* caluse.

Query 5.1.1

```
select *
from R
where exists ( select * from S where S.b = R.a )
```

There are two main optimization techniques employed by relational (and object-oriented) database systems for improving the performance of queries with correlated subqueries: (1) "folding" a subquery into the outer query via query rewrite prior to optimization, and (2) caching subquery results. Each technique may be preferred in different situations. Our work adapts and extends the relational subquery result caching technique for queries over semistructured data.

In a relational system the caching mechanism can be fairly straightforward. Consider Query 5.1.1. Assume that the optimizer chooses not to fold the subquery into the outer query and that no index on `S.b` exists. Then, the most obvious way to execute this query is to reevaluate the subquery for every tuple in `R`. This approach can be very inefficient since it effectively introduces a cross-product. A very simple one-element "cache" could be introduced to remember the most recent `R.a` value seen, and whether or the not the existential predicate was satisfied. Then the subquery is reevaluated only when a new `R.a` value is seen. This simple cache can improve performance considerably when there are many duplicate `R.a` values and the tuples of `R` are fetched in sorted order on the `a` attribute. The obvious generalization of this approach uses a fixed-size cache with many ⟨`R.a`, *predicate-result*⟩ pairs, in which case we expect a performance improvement even when the `R.a` values are not sorted.

We propose a more general technique that caches results from portions of a physical query plan. The idea is similar to caching results from a correlated subquery in the relational model: results from a subplan can be cached and reused when the information that the subplan depends on has not changed. Our technique is called *subplan caching*. The technique identifies subplans that are likely to benefit from a fixed-size in-memory cache, and inserts new *Cache* query operators accordingly. We describe extensions to Lore's query engine to implement our technique. Because the plan space searched by Lore's optimizer is expanded considerably by this new technique, we describe heuristics that prune the additional search space to a manageable size, and a cost-based approach that chooses the cheapest plan from the new space.

The remainder of this chapter proceeds as follows. We motivate our subplan caching technique using examples in Section 5.2. Preliminary information required to understand the optimization technique is given in Section 5.3. The wide range of scenarios in which subplan caching can be applied are shown in Section 5.4. In Section 5.5 we describe our generic caching mechanism, encapsulated in a physical query operator that can be inserted anywhere in a query plan. In Section 5.6 we describe heuristics and a cost-based mechanism for deciding which subplan results to cache. We report experimental results of our implementation in Section 5.7. Related work is presented in Section 5.8.

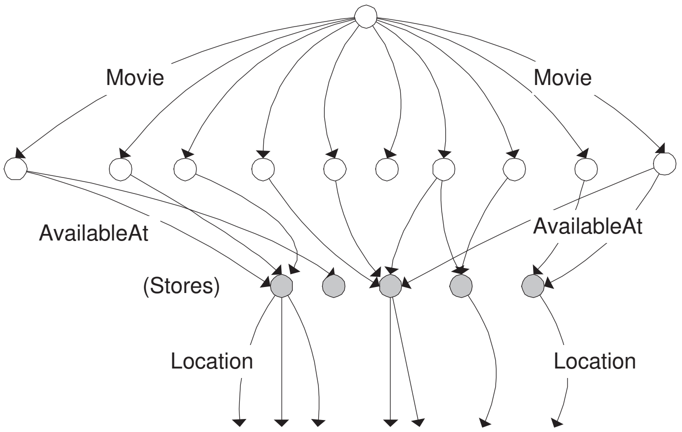

Figure 5.1: Some of the paths from the Movie Store database

5.2 Motivating Examples

There are many situations where introducing a small in-memory cache can decrease query execution time considerably. In general, the use of a cache avoids reexecution of a subplan, thus decreasing the total amount of disk I/O. Since subplan caching operates at the granularity of subplans, it is strictly more general than the subquery result caching discussed in Section 5.1: All subqueries become subplans during query plan generation, but subplans often do not correspond to subqueries.

As an example of when a cache is useful, consider our Movie Store database from Chapter 2 (Section 2.3, Figure 2.4). Recall that this database contains information about movies, stores that rent and sell the movies, companies that own the stores, and people that work for the companies or have participated in making a movie. In Figure 5.1 we summarize a portion of the general shape of the Movie Store database.

Example 5.2.1 Consider Figure 5.1 and the paths through the store objects (shaded in the figure). Notice that the store objects have many paths that feed into and feed out of them, however the set of store objects is fairly small. Suppose a simple top-down execution strategy (recall Chapter 3) is used to discover store locations by matching path expression `Movie.AvailableAt.Location`. This strategy results in store locations being revisited many times, once for each way a store is reached. A small cache to remember the locations associated with given store objects would be beneficial. □

Example 5.2.2 Given the same path expression, "`Movie.AvailableAt.Location`", and the same database in Figure 5.1, consider a bottom-up execution strategy. Again, store objects will be bound many times, since a store can have many locations and bottom-up query execution will visit a store object once for each location. Therefore, the subpaths `Movie.AvailableAt` above a store object will each be traversed many times. A cache to remember the `Movie.AvailableAt` paths above store objects would be beneficial. This same argument would hold even if the database were tree-structured, i.e., even if there were only one `Movie.AvailableAt` path to each store object. □

5.3 Preliminaries

Recall from Chapter 3 that a single logical query plan is generated from the parsed query, and then the space of physical query plans is searched. The cost-based optimizer selects the physical query plan with the smallest estimated cost. A *subplan*, s, of a physical query plan is identified by a single physical node, n, and includes all of n's descendants. (Physical query plans are always trees.) We say that s is *rooted* at n. For example, Figure 5.2(a) contains one possible physical query plan for the following query. This query is intended for the Movie Store database. It finds all ⟨title, director⟩ pairs for comedies.

Query 5.3.1

```
select t, d
from MovieStore.Movies x, x.Movie m, m.Title t, m.Director d
where exists g in m.Genre: g = "Comedy"
```

We have isolated the subplan rooted at the upper *Select* node from Figure 5.2(a) in Figure 5.2(b).

In order to determine if it is advantageous to add a cache operator over a subplan, we must identify two sets of variables, the *dependent set* (DS) and *provides set* (PS), which are properties of every subplan. In Section 5.6 we will discuss how these variable sets are used in the subplan caching technique. The two variable sets, DS and PS, are defined as:

- The *dependent set* (DS) for a subplan is the set of all variables that must be bound by a physical operator executed before the subplan. DS $= I - O$, where I is the set of all variables that are required by some physical operator in the subplan as "input", and

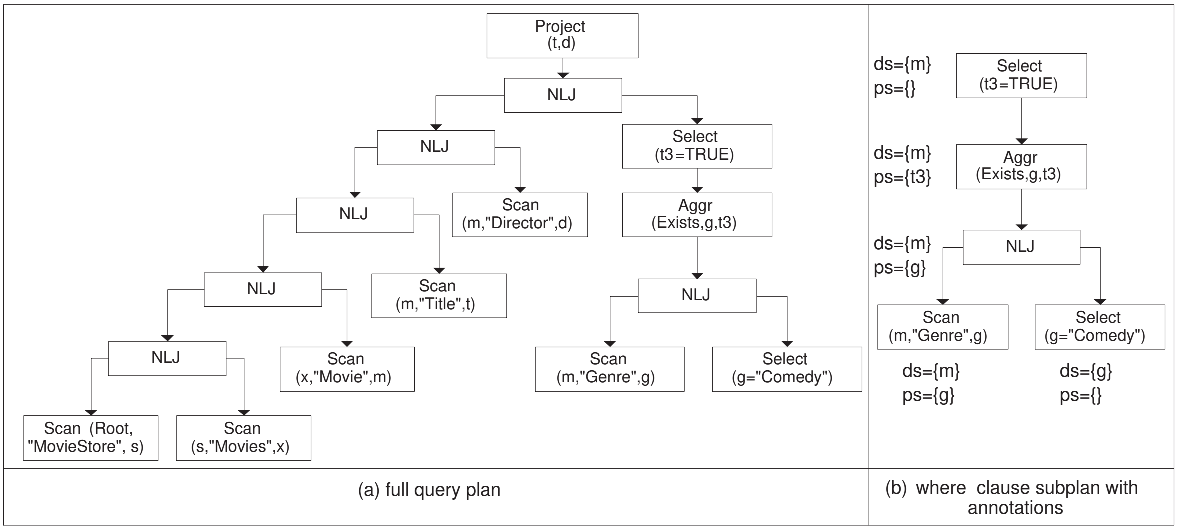

Figure 5.2: A sample physical query plan for Query 5.3.1

O is the set of all variables that are generated as output by some physical operator in the subplan. For example, in Figure 5.2(b) the topmost node has DS $\{m\}$ since the input variable set is $\{t3, m, g\}$ and the output variable set is $\{t3, g\}$. m is bound by a *Scan* operator that appears in the complete plan of Figure 5.2(a).

- The *provides set* (PS) for a subplan is the set of variables that are bound by the subplan and used after the subplan. In Figure 5.2(b) the PS for the entire subplan is empty since no variables that are bound by the subplan are required later. In Figure 5.2(a) the PS for the lowest-left *NLJ* operator is $\{x\}$. The PS for a subplan is computed by intersecting the set of all variables that are generated as output by the subplan with the input variables of all operators that execute after the subplan.

Figure 5.2(b) is annotated with the DS and PS sets for each subplan. The DS and PS for subplans are used to determine good placement of *Cache* operators and are used directly in the execution of the *Cache* operator.

The *Cache* operator must be flexible enough so that it can be placed on top of any subplan. Logically the operator contains a fixed-size cache of $\langle k, d\rangle$ pairs, where k is an evaluation for variables in the DS, and d is the set of evaluations for variables in the PS. k is the cache *lookup element*, or key, and its values are unique in the cache. d is the associated (or secondary) data. In our context d is the set of evaluations that are a result of the *Cache*'s subplan when k is bound to the variables in DS.

5.4 Subplan Caching Examples

The subplan caching technique identifies subplans that are expected to execute many times but with few distinct bindings for variables in the DS, and thus are expected to benefit from our caching techniques. In Section 5.2 we described in very general terms two scenarios where query execution would benefit from the subplan cache optimization. In this section we describe in detail, using specific physical query plans and placement of `Cache` operators, three scenarios where subplan caching is useful.

Example 5.4.1 Consider execution of the plan shown in Figure 5.2(a) for Query 5.3.1. All *NLJ* operators in Figure 5.2(a) are dependent joins. Recall from Chapter 3 that dependent joins do not contain explicit join conditions and pass bindings from the left side of the *NLJ* to the right side. Notice that the DS for the subplan corresponding to the *where* clause, shown in Figure 5.2(b), only contains the variable m. However, this subplan will be called once for every valid set of bindings for m, t, and d.[1] By introducing a *Cache* operator above the subplan for the *where* clause we prevent reexecution of the clause when the m binding already appears in the cache. Notice that in this situation the cache hit ratio will be very high (even with a tree-structured database) since m will be bound to the same object for each $\langle t, d \rangle$ subobject pair of m. Recall that in the Movie database a movie can have several directors and title subobjects. □

Example 5.4.2 Another example where subplan caching can be beneficial is when the *Cache* operator is placed directly above a single (leaf) physical operator. For example, in Figure 5.2(a) a *Cache* operator directly above the `Scan(m.Director d)` operator would be beneficial in two situations:

1. If we expect a movie to have many different `Title` subobjects, then rediscovering the `Director`'s for each binding of the t variable would be wasteful. We can cache the set of `Director`'s for a movie to avoid searching for and fetching the directors again.

[1] Another optimization, which we do not explore in this thesis, moves the operators that bind t and d to after the *where* clause, ensuring that we only look for `Title`'s and `Director`'s after we know that the movie satisfies the *where* clause. It is clear that subplan caching is applicable in a wider number of situations than such an optimization.

2. Suppose that `Movie` objects are reachable via many paths and m is bound to the same object many times. Then we can cache the set of directors for a movie. While this scenario is unlikely given the simplicity of Query 5.3.1, this situation can often arise for graph-structured databases. □

Example 5.4.3 Finally, a third example where the *Cache* operator is beneficial involves subqueries, which appear fairly frequently in Lorel. Consider the following query which fetches reviewers who scored a movie higher than the average review score for that movie.

```
select r
from MovieStore.Movies x, x.Movie m, m.Reviewer r
where r.Score > Avg(m.Reviewer.Score)
```

A typical plan would have a subplan for `Avg(m.Reviewer.Score)` with a DS of $\{m\}$, while the complete *where* clause has a DS of $\{r, m\}$. Intuitively, the average reviewer score for a movie m need not be recomputed for each reviewer r of that movie. A *Cache* above the plan for the subquery allows the aggregate value to be reused when m is in the cache. □

5.5 The Cache Physical Operator

The *Cache* operator can be placed over any subplan s and is parameterized by three main properties:

1. The DS for s, which describes the cache lookup key.
2. The PS for s, which describes the secondary data that is tracked in the cache.
3. The amount of memory allocated to the cache.

Physically, each cache entry in the *Cache* operator is represented using a structure similar to the evaluations within an encapsulated evaluation set (EES) introduced in Chapter 3 (Section 3.4.2). The one difference between the internal structure in the *Cache* operator and the evaluations in an EES is that multiple primary variables are allowed: The primary variables correspond to the set DS, and the secondary variables correspond to the set PS.

The function of the *Cache* operator is as follows. A request is made for the *Cache*'s next evaluation by the parent operator of the *Cache*, which provides bindings for the DS. The

Cache operator probes the in-memory cache based DS binding. If a match is found, then the associated set of evaluations for the PS are extracted from the cache. The first of the PS bindings are added to the current evaluation and returned to the parent of the cache operator without executing the child. Since there are a set of PS bindings for a single DS, subsequent requests to the *Cache* operator with the same DS bindings result in the next element in the set of PS evaluations to be passed up to the parent. If the DS evaluation is not found in the cache then s is instructed to get its next evaluation. The bindings for the variables in the PS for the returned evaluation are stored with the key element and the next evaluation is requested from s. The procedure repeats until s indicates that no more evaluations exist for the bindings of the DS. The key and secondary evaluations are added to the cache and the procedure continues as if a cache hit had originally occurred.

Like any cache data structure, the *Cache* physical operator must support two main operations efficiently: *cache lookup*—fetching a cache entry based on a key value—and *victim selection*—selecting a cache entry to remove when the cache is full and a new entry is being added. While cache lookups occur much more frequently than victim selection, both operations must be supported efficiently.

Cache lookup is supported efficiently in the *Cache* operator by a hash table keyed on the cache key. We experimented with two different victim selection algorithms. In the first algorithm, our least-frequently-used (LFU) algorithm, we use a counter to track how many times an entry has been accessed. Victim selection deletes the cache entry with the lowest count in the same hash bucket that the new element will occupy. If a bucket contains no elements then a scan of the entire hash table is used to find the entry with the lowest count. Our second victim selection algorithm uses a variation on the *second chance* algorithm [SG98]. A *reference bit* is associated with each cache entry. A cache entry's reference bit is set to 0 initially, and is reset to 0 whenever it is referenced. During victim selection the hash table is scanned and any cache entry whose reference bit is 0 is set to 1. When a cache entry has a bit already set to 1 then that entry is chosen as the victim. Experiments in Section 5.7 show that the two victim selection algorithms perform similarily.

5.6 Placement of the Cache Physical Operator

The cache managed by a *Cache* operator lives in memory and does not, by itself, access the disk. Thus, inserting a *Cache* operator in the query plan cannot increase I/O. However, the overhead associated with the cache does increase CPU time so poor placement of a *Cache* operator can increase overall query execution time. We discuss two methods that can be used to determine where to place *Cache* operators, then describe our approach, which combines the two methods.

5.6.1 Heuristic Placement

Simple heuristics can be used to predict when it may be advantageous to add a *Cache* operator. Examples of placement heuristics include:

1. Don't use the *Cache* operator in conjunction with certain physical operators and locations in the query plan. For example, in a majority of cases it wouldn't be advantageous to place a *Cache* operator directly over a *Sort* operator, since *Sort* is usually executed a single time and creates a temporary result of its own. Similarily, a *Cache* operator over the left subplan of a NLJ is unlikely to help, especially in left-deep trees where the DS is empty.

2. If the set of PS evaluations for each DS evaluation is estimated to be large then the cache will likely fill up very quickly with few cache entries. Also, if DS contains many variables then the chances of a cache hit can decrease (since the combinations of objects assigned to multiple variables is commonly larger than combinations of objects assigned to a smaller number of variables). Therefore, one heuristic is to only use a *Cache* operator when the DS contains fewer than d variables (e.g., $d = 2$) and the PS contains fewer than p variables (e.g., $p = 4$).

3. We could add a *Cache* operator only when the total predicted size of potentially cached data (the estimated size of the PS evaluations for the estimated number of DS evaluations) is less than some factor of the size allocated to the cache.

5.6.2 Cost-based Placement

An alternative to heuristic placement of *Cache* operators is cost-based placement. Considering all possible placements of *Cache* operators for all possible plans is infeasible, so an alternative possible cost-based placement is as follows. Recall from Chapter 3 that the Lore optimizer creates a physical query plan in a top-down fashion, with each logical query plan node responsible for creating the optimal physical query plan for the subplan rooted at that node. We could extend each logical query plan node to create the optimal physical plan and then cost two separate alternatives: one with the optimal plan, and one that places a *Cache* operator above the optimal plan. This heuristic reduces the number of placements considered since the placement of a *Cache* operator over subplan s does not affect the placement of *Cache* operators within s. Also, some logical query plan nodes are translated into more than one physical query plan node and the placement of the *Cache* operator is only considered a single time. For example, existential quantification in the logical plan is a single operator, but in the physical query plan it can consist of two physical operators.

5.6.3 Combination of Heuristic and Cost-Based Placement

The *Cache* operator placement technique we use combines the heuristic and cost-based approaches. We use heuristics (1) and (2) from Section 5.6.1 along with the cost-based proposal in Section 5.6.2. More specifically, the *Cache* operator will only be considered above a subplan s when the following conditions hold:

- s is the right child of a *NLJ* physical operator, a subquery, an aggregated result (rooted by an *Aggr* physical operator), or an arithmetic operation (rooted by an *Arith* physical operator).
- The DS for s contains a single variable and the PS contains fewer than four variables.

When the above conditions are met, then during plan generation a *Cache* physical operator is placed above s and its cost is compared with the cost of s without the *Cache* physical operator.

As we will show in Section 5.7.6, poor placement of *Cache* operators can adversely affect query performance. To estimate the cost of a *Cache* operator we use two terms: the predicted number of distinct objects bound to the DS variable, *disObj*, and the estimated number of times that the *Cache* subplan will be asked for its next evaluation, *numCalled*.

For *disObj*, the distinct object count for the DS variable, the *Cache* operator makes use of the set of path expression components, P, that are bound by physical operators that execute before the *Cache* operator. There are two cases. In the first case, the DS variable appears as a source variable in an element of P. In this case the plan is involved in a reverse evaluation of P and the distinct object count is the number of distinct objects that *begin* a path. In the second case, the DS variable appears as the destination variable in an element of P. Thus, a forward evaluation is in progress and the distinct object count is the number of distinct objects *ending* a path. In both cases statistics tracked by the system ($|p|_d$ and $|p|^d$ from Chapter 3, Section 3.4.3) provide the distinct count. *numCalled*, the estimated number of executions of the *Cache* subplan, is determined by the formulas in Chapter 3 (Table 3.7). These formulas use the statistics and estimated number of results from the physical operators that execute before the *Cache* operator.

Dividing *disObj* by *numCalled* results in the approximate percentage of distinct object bindings that will be fed into the *Cache* subplan with respect to the number of times the subplan will be called. In determining the cost of a *Cache* subplan, we do not attempt to model the behavior of the cache, but instead assume (optimistically) that cache elements remain in the cache until they are no longer needed. There are an estimated *numCalled* requests made to the *Cache* operator. For *disObj* of these calls the cost of executing the *Cache* is the cost of executing the *Cache*'s subplan along with some overhead introduced by the cache. For the remaining (*numCalled*−*disObj*) calls to the *Cache* operator, the cost is the CPU overhead for looking up the cache key and data elements.

The predicted I/O cost of the *Cache* subplan is always less than or equal to the I/O cost of the subplan without the *Cache* operator, but the CPU cost of the *Cache* subplan is always higher. Recall from Chapter 3 (Section 3.4.3) that I/O cost is the determining factor for selecting query plans in Lore and CPU cost is used only as a "tie-breaker". To avoid selecting the *Cache* subplan when the decrease in I/O cost is very small (and to informally factor in the increased CPU cost of the *Cache* operator), we only select the *Cache* subplan when the I/O cost is estimated to be 20% less than the I/O cost of the subplan without the *Cache* operator. While 20% worked well in our experiments for a variety of databases and queries, a better solution, not considered further in this thesis, is to integrate CPU and I/O estimates more thoroughly in our cost model. Note that to do so requires detailed knowledge of the CPU speed, disk seek time, disk latency time, and disk transfer time.

5.7 Experimental Results

We have implemented the subplan caching technique in the Lore system, with a switch that allows us to optimize a query with and without subplan caching. We call the plan created when subplan caching is active the *SC plan* and the plan created by the Lore optimzer without subplan caching the *normal plan*. We used three different databases to

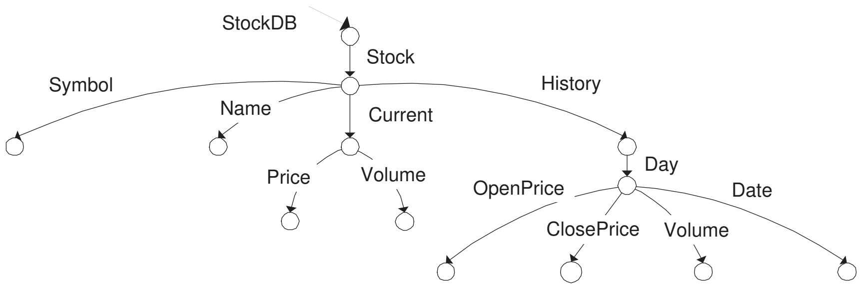

Figure 5.3: DataGuide for the StockDB database

test the subplan caching technique: The Database Group database (Chapter 2, Figure 2.2) consisting of 3,633 objects, the Movie database (Chapter 2, Figure 2.3) consisting of 62,256 objects, and the *StockDB* database, introduced in this section in Figure 5.3 and consisting of 11,298 objects. The StockDB database is a fairly regular tree of depth up to five containing data about stocks. We experimented with both victim selection algorithms presented in Section 5.5 and found that their performance was similar. Unless otherwise stated the size of each in-memory cache was limited to 4K and the LFU victim selection algorithm was used.

In all of the experiments below we created and executed both normal and SC plans. In all experiments the normal plan and the SC plan were exactly the same except for the inclusion of one or more *Cache* operators in the SC plan. It is possible to envision situations where the SC plan and the normal plan are completely different, however we found that most of these situations do not have database shapes and queries that occur naturally.

Some of the queries in the experiments reported below can be written in a shorter or more natural form. For example, we have chosen to increase the length of some path expressions in order to test all aspects of the subplan caching technique.

Optimization time for the subplan caching technique. It is important that the time saved during query execution exceed the time required to perform additional optimization. In order to gauge the amount of time that the subplan caching technique adds to query optimization we ran 13 experiments over all three databases with queries of varying complexity. On average the subplan caching technique introduced a 10% increase in optimization time. In all cases the increase in optimization time was much smaller than the amount of query execution time saved by the new technique.

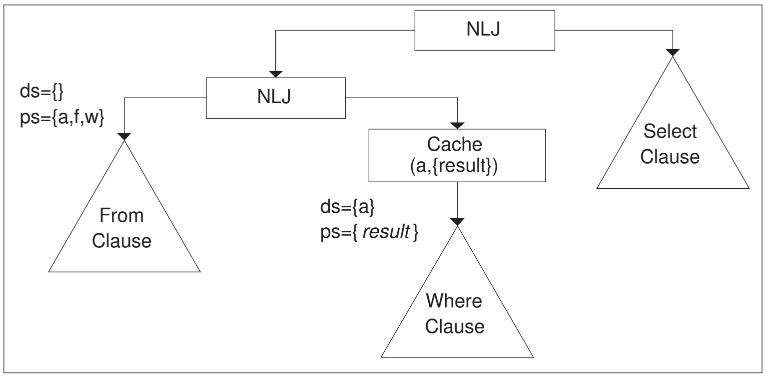

Figure 5.4: Structure of the query plan for Experiment 5.7.1 with subplan caching

Experiment 5.7.1 (Caching a subquery) The query in this experiment is executed over the Movie database. This query fetches the names of all actors who have appeared in more than two movies and the movie titles that they have appeared in. The query is:

```
select a.Name, w.Title
from Movies.Movie m, m.Actor a, a.Film f, f.Movie w
where count(a.Film.Movie) > 2
```

The path expression `Movies.Movie m, m.Actor a` retrieves all people who have acted in at least one movie. The subpath `a.Film f, f.Movie w` points from each of the actors to the set of films that they acted in.

The normal plan created by Lore's optimizer uses a top-down query execution strategy. When subplan caching is enabled a *Cache* operator is placed directly above the subplan for the *where* clause in the top-down plan. This *Cache* operator caches the result of the *where* clause for a given object bound to a. The overall structure of this query plan is shown in Figure 5.4. In this figure and those that follow, the *Cache* operator has two parameters, DS (a single variable) and PS (a set of variables). In this experiment the *Cache* operator improves the performance of the normal plan in two ways. First, when an actor appears in more than one movie he will be bound multiple times to a. Second, for a single binding for a, different objects may be bound to f and w. The SC plan executed in 17.158 seconds while the normal query plan executed in 28.621 seconds. □

Experiment 5.7.2 (Caching a subplan) The query in the second experiment was executed over the StockDB database, whose DataGuide is shown in Figure 5.3. The query

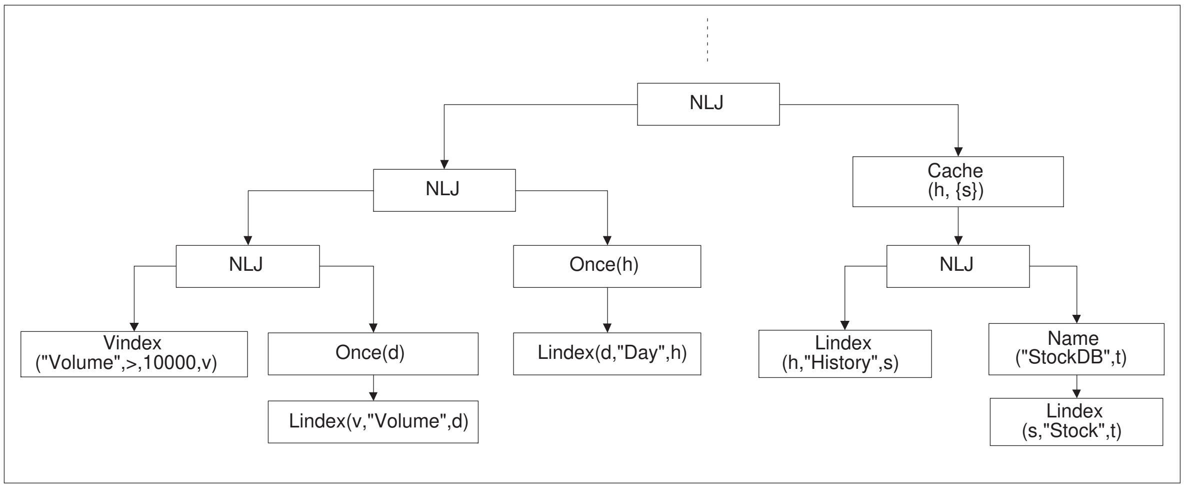

Figure 5.5: Query plan for Experiment 5.7.2 with subplan caching

returns a stock symbol once for each time that stock had a high volume of trade in the past.

```
select s.Symbol
from StockDB.Stock s, s.History h
where h.Day.Volume > 100000
```

Without subplan caching the optimizer chose a bottom-up query execution strategy since the query contains a fairly selective predicate in the *where* clause. The details of a portion of the plan for this query are shown in Figure 5.5. It is a bottom-up plan: A *Vindex* is used to satisfy the predicate, and *Lindex* operators traverse up through the tree to ensure that the object found by the *Vindex* satisfies the path in the database. Subplan caching placed a single *Cache* operator above the *Lindex* operators that satisfy the subpath "`StockDB.Stock s, s.History h`" (shown in Figure 5.5) to avoid having to match this subpath multiple times when a stock was traded at a high volume on more than one occasion. That is, if a stock had a high volume of trade on many days then the cache can avoid having to match up the subpath "`StockDB.Stock s, s.History h`" many times. The SC plan executed in 0.201 seconds while the normal plan took 0.55 seconds. □

Experiment 5.7.3 (Multiple Cache operators) This experiment shows how multiple *Cache* operators can be useful within a single query plan. The following query is executed

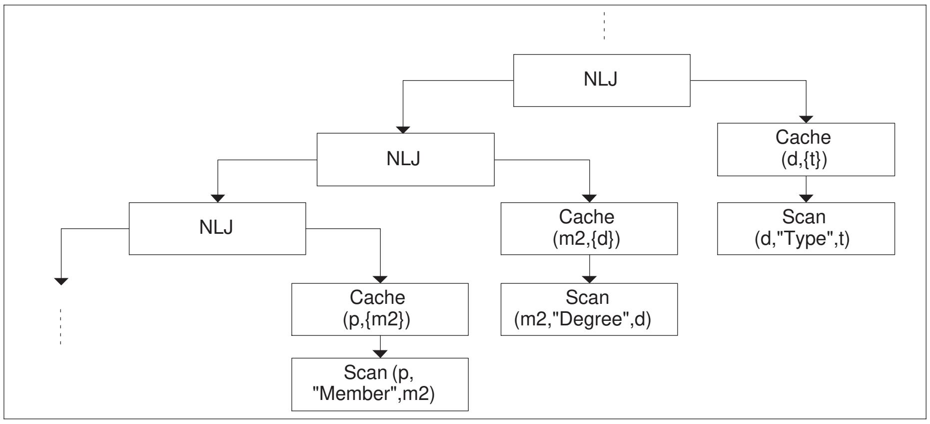

Figure 5.6: Several *Cache* operators in a single plan

over the Database Group database. The query retrieves all project name, degree type pairs in the database such that at least one group member works on that project and has the degree type:

```
select p.Name, t
from DBGroup.Member m, m.Project p,
       p.Member m2, m2.Degree d, d.Type t
```

A top-down plan is generated by the optimizer without subplan caching. When subplan caching is enabled three *Cache* operators are placed in the top-down plan. A portion of the SC plan is shown in Figure 5.6. The *Cache* operators decrease query execution time because there are few distinct projects in the database, but many group members who work on these projects. Instead of rediscovering all the `Member`'s and then `Degree`'s and `Type`'s for those members, a *Cache* is used. Each *Cache* operator has a PS consisting of a single variable corresponding to the variable bound in the *Scan* operator that it appears directly above. The query execution time for the SC plan is 10.566 seconds, while the query exection time for the normal plan is 15.213. □

Experiment 5.7.4 (Nested Cache operators) *Cache* operators can be useful even when one *Cache* appears within the subplan of another. Consider the following query, executed over the Database Group database, that retrieves the names of projects and the set of names for the group members that work on that project:

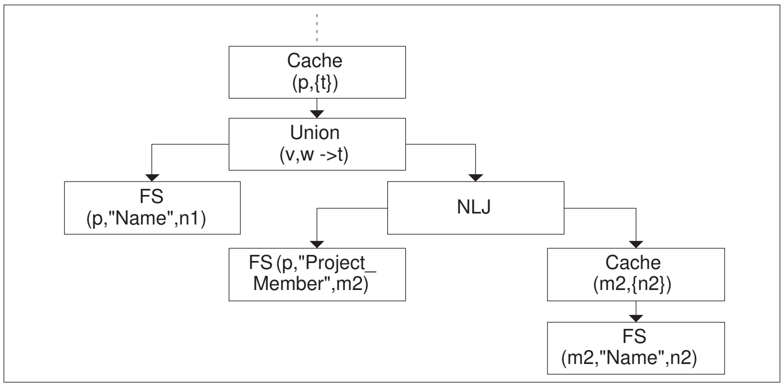

Figure 5.7: Nested *Cache* operators

```
select p.Name, (select n2 from p.Member m2, m2.Name n2)
from DBGroup.Member m, m.Project p
```

Without subplan caching the optimizer constructed a top-down plan for this query. When subplan caching is enabled the top-down plan is augmented with two *Cache* operators, both in the subplan responsible for executing the *select* clause. The first *Cache* operator is placed above the subplan for the outermost *select* clause. The second *Cache* operator caches the `Name` subobjects of a project member. The relevant portion of the plan is shown in Figure 5.7. The first *Cache* operator caches the results of the *select* clause for each project and improves query execution time because the same project may be bound to p many times. The second cache operator caches names of project members, since people typically work on more than one project. The SC plan executes in 7.4711 seconds versus 10.2155 seconds for the normal plan. □

Experiment 5.7.5 (Varying the size of the cache) One obvious factor that influences the performance of a plan containing a *Cache* operator is the amount of memory allocated to the cache. In the following query, executed over the Database Group database, a regular expression operator finds group members who have an advisor who is connected (in some way) with semistructured data. We use the `#[4]` operator to search four levels deep (following any path) to bind the variable s. Recall that # is preprocessed to be $(.l_1|...|l_n)^*$ where $l_1...l_2$ is the set of labels in the database.

```
select m
from DBGroup.Member m, m.Advisor a, a.#[4] s
where s = "Semistructured data"
```

In a normal top-down evaluation of the query the regular expression operator would be evaluated many times even though the number of distinct objects bound to a is small (since advisors have many students). The optimizer with subplan caching created a top-down query plan with a single *Cache* operator directly over the `Scan (a.#[4] s)`. This *Cache* operator has a DS=$\{a\}$ and a PS=$\{s\}$. Obviously, this cache can be useful to avoid reexecution of the regular expression. However, the number of objects that satisfy the regular expression, those bound to s, is large, so each cache entry is large. In fact, for this query and database a small cache will hold at most one or two cache entries. We varied the size of the cache from 4K to 64K to observe the difference in query execution time. The results are shown in Figure 5.8. Without a cache the query takes over 30 seconds to execute. The SC plan executed between 9 and 22 seconds faster than the normal plan. Notice that the increasing size of the cache does not result in a linear improvement in the query execution time. In fact, from 4k to 32k the amount of time increases slightly, because the increased cache size did not result in a higher cache hit ratio and the larger sized cache incurred slightly higher maintenance costs. □

Experiment 5.7.6 (Experiment 6—Poor placement of the Cache operator) As discussed earlier, it is important that the *Cache* operator be placed judiciously. For this experiment we tried both victim selection algorithms, LFU and our modified second-chance algorithm, discussed in Section 5.5. Consider the following query executed over the Movie database:

```
select m
from Movies.Actor a, a.Film f, f.Movie m
```

This query retrieves all movies that at least one actor appeared in. Our decision procedure will correctly not place any *Cache* operators in a top-down evaluation of this query. To illustrate the performance penalty possible with indiscriminate placement we forced two *Cache* operators to be placed: one to cache the actor objects and the other to cache film objects. These are poor placements since all of the objects bound to a and f are unique, resulting in a cache hit rate of 0%. The impact of the two cache operators is shown in

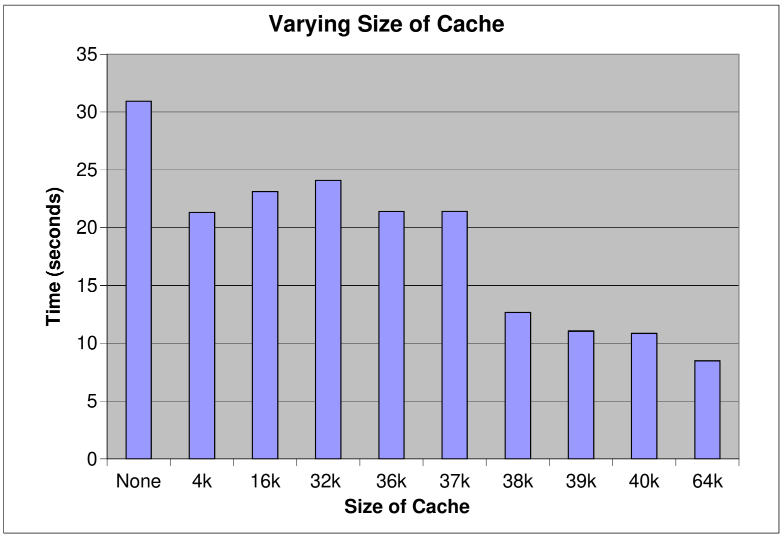

Figure 5.8: Varying the size of the cache

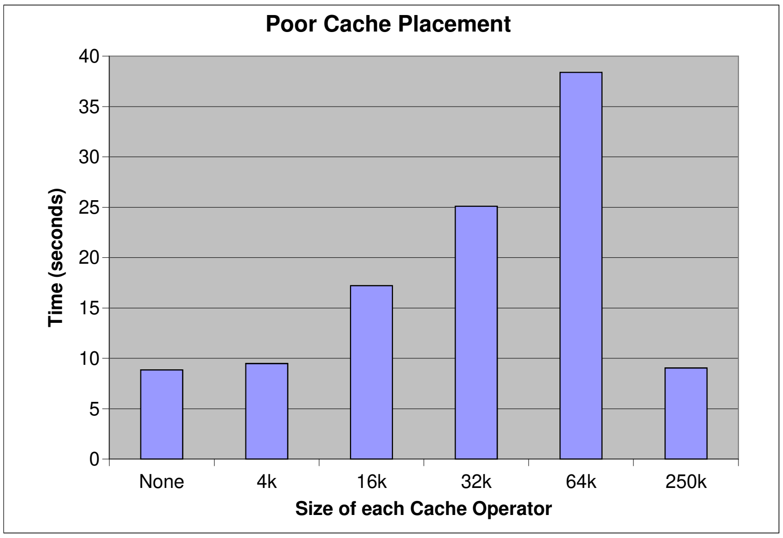

Figure 5.9: Poor placement of several cache operators with varying cache size

Figure 5.9. The running time for LFU victim selection was a bit longer than our second-chance algorithm for smaller cache sizes, but almost the same for cache sizes above 32k. As shown in Figure 5.9, the top-down query plan without any cache operators executes in just under 9 seconds. The two cache operators add a minimum of 0.5 seconds to the execution time. As the size of the memory allocated to each *Cache* operator increases, the overhead associated with maintaining the cache and choosing victim cache elements also increases. For a cache of size 64k the query execution time has more than tripled. It isn't until the cache has size 250k or more that it can hold all of the entries and no movement out of the cache need occur. Then the overhead associated with the cache is very small and the query execution time falls to just above 9 seconds. □

5.8 Related Work

Caches in query plans have been considered as far back as the original "Access Path Selection" paper [SAC$^+$79], where a brief mention is made about avoiding the reevaluation of a subquery when the current referenced attributes are the same as those in the previous candidate tuple. More recently, [RR98] considered reusing "invariants", or portions of a correlated subquery that do not change when the outer bindings change. Their optimization centers around subqueries, while we consider the broader application of subplans.

In [YM98] there is a brief mention of the usefulness of caching objects during long path traversals. The authors state that a caching technique would be "especially effective if the path to be traversed is long" ([YM98] Page 66). This observation is a good argument for our optimization technique.

Chapter 6

Optimizing Path Expressions

Path expressions, introduced in Chapter 2, play an important role in the Lorel query language. In Chapter 3 we introduced the general framework of Lore's query optimizer, which handles arbitrary path expressions within the context of a complete Lorel query. In this chapter we focus exclusively on optimizing complex path expressions, introducing two techniques beyond those in previous chapters. The first technique explores a variety of algorithms to create a physical query plan for path expression evaluation, each algorithm using different heuristics and physical plan search strategies. The second technique is a post-optimization step that introduces a grouping operation at certain points in the physical query plan for a path expression, improving overall efficiency of the plan.

6.1 Introduction

Path expressions play a key role in the Lorel query language, and in all query languages for semistructured data. The original Lore query engine, described in Chapter 3, generates plans for all path expressions using the same plan generation algorithm, and does so in the context of optimizing a full Lorel query. This approach, along with the original pruning heuristics (Chapter 3, Section 3.4.4), resulted in the following limitations:

- Locally optimal decisions were made that did not always result in globally optimal plans.
- Only a subset of all possible path expression component orderings was considered, and this subset depended on the order in which the user specified the path expression

components.

- When a branching path expression (recall Chapter 2, Section 2.4.7) appeared in the query, no attempt was made to distinguish between components of the path expression that explored different portions of the database.

Besides these limitations, the original Lore optimizer did not take advantage of fairly common database "shapes" that can benefit from different optimization techniques.

In this chapter we focus on two new optimization techniques designed specifically for path expressions. The first optimization, appearing in Section 6.2, could replace certain portions of the Lore physical query plan enumerator. Recall from Chapter 3 (Section 3.4.4) that the query plan enumerator consists of a physical query plan search strategy, along with heuristics that prune the search space. In Section 6.2 we introduce several different algorithms that can replace both plan searching and pruning heuristics in the Lore optimizer, for the special case of path expressions. The second optimization, appearing in Section 6.3, introduces a post-optimization technique that can be applied to a physical query plan created by the Lore optimizer. The technique identifies path expressions that are expected to match many paths through the data, but the paths at some time pass through a small set of objects. In these situations duplicate work can be avoided by creating an EES (recall Chapter 3, Section 3.4.2) at appropriate points in the plan. Related work for both optimization techniques is presented in Section 6.4.

6.2 Branching Path Expression Optimization

Recall (from Chapter 3, Section 2.4.7) that a *branching path expression* is a path expression containing at least one variable that appears as the source variable in more than one component of the path expression. As a simple example of a branching path expression, consider the *from* clause of the following query, which finds the names of movies along with the names of actors that appeared in sequels and prequels of a movie. This query is intended to be executed over the Library database given in Chapter 2 (Section 2.3) and shown in Figure 2.5.

```
select n1, n2, n3
from Library.Movies s, s.Movie m, m.Name n1,
                                  m.Prequel p, p.Actor a1, a1.Name n2
                                  m.Sequel s, s.Actor a2, a2.Name n3
```

The branching path expression in the *from* clause of this query explores both the prequel and sequel subgraphs of a movie object. The original Lore optimizer, described in Chapter 3, will produce the best physical query plan for this query that is within the search space that the optimizer considers. However, due to pruning heuristics (recall Chapter 3, Section 3.4.4), the optimizer will not attempt to reorder the execution of the path expression components so that "`m.Prequel p`" and "`m.Sequel s`" are executed one after the other and before the set of all actors and names are discovered for either branch. The advantage in first discovering both `Prequel` and `Sequel` subobjects for a movie is that only for those movies that have both will query execution go on to fetch all the actors and their names for both the prequel and sequel movies. In fact, no query plan produced by the optimizer described in Chapter 3 would first find the movies that have both prequels and sequels before satisfying the remaining path expression components: "`p.Actor a1, a1.Name n2, s.Actor a2, a2.Name n3`".

Due to its pruning heuristics, the original Lore optimizer could not reorder the execution of path expression components as shown in the previous example. In this section we consider a variety of algorithms that can reorder path expression components to various extents. In general, when an algorithm considers more reorderings the algorithm is much slower, but it may result in a much better plan. In some of our algorithms all reorderings are considered. In others, we specifically restrict reorderings based on the branching structure of the path expression. For example, we may considering reordering entire branches of a path expression, but not reordering components within a branch. Of course all of our algorithms apply to the special case of path expressions without branches.

The optimizations presented here focus on the evaluation of path expressions only. As described in Chapter 3 (Section 3.4.1), it is the responsibility of the *Chain* logical operator to optimize entire path expressions, and a long path expression results in a series of nested chain operators. The algorithms presented here could be used in a *Chain* operator at any level, replacing the previous search strategy and pruning heuristics for the path expression rooted at that operator. However, we do not explore the complete integration of the optimizations presented in this chapter into the Lore optimizer.

The contributions of this section are:

- We present six algorithms that reduce the search space of possible query plans for branching path expressions. Our algorithms reduce the search space (as defined by the physical operators given in Chapter 3, Section 3.4.2) in different ways and to different extents. We include among the six algorithms the algorithm to optimize path expressions that results from the original Lore optimizer as described in Chapter 3.
- We introduce four post-optimization transformations that can be applied to the query plan for a path expression. The post-optimizations move either entire branches of a path expression or individual components to more advantageous positions in the plan.
- Each algorithm and post-optimization has been implemented using the Lore infrastructure, and we present experiments showing their strengths and weaknesses. In the experiments we compare optimization and execution times across the different algorithms, and for small queries we compare their times against the optimal plan produced by an exhaustive search of the plan space for branching path expressions.

6.2.1 Preliminaries

We will present several algorithms that produce a physical query plan for a given branching path expression. A list of path expression components, s, is provided as input to each algorithm, and the output is the optimal plan within the search space for that algorithm. In all of our algorithms the list s may be an arbitrarily complex branching path expression. Recall from Chapter 2 (Section 2.4.7) that a path expression component is a triple $\langle$*source variable*, *subpath*, *destination variable*$\rangle$, often denoted in this chapter as "`x.subpath y`" where x and y are the source and destination variables, respectively. A path expression component in s may contain a subpath with regular expression operators, although the techniques presented here are not designed specifically to handle general path expressions in the most efficient way; optimization techniques for general path expressions were explored in Chapter 4. In some situations it is necessary to isolate the individual "branches" in s. We construct a set, r, containing the individual branches. Specifically, r is a set of lists of components created from s such that:

1. Each component in s appears in a single list in r, and each list in r contains components only found in s.

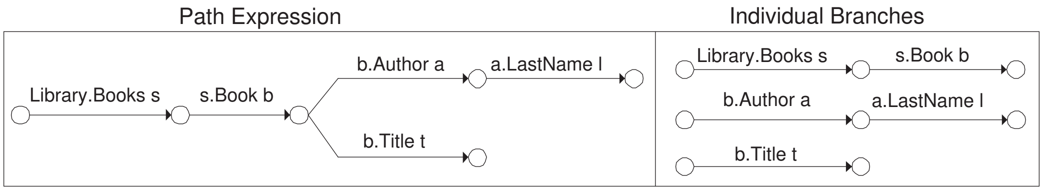

Figure 6.1: A branching path expression

2. Each list in r specifies a *linear path*: each component's destination variable appears as the source variable for the next component in the list (except the last).

3. If a source variable is used in more than one component in s, then each component with that source variable starts a new list in r.

4. It is not possible to combine two lists of r without violating (2) or (3).

It is easy to construct r in time linear in the length of s. As an example of the decomposition, suppose s = ⟨`Library.Books s, s.Book b, b.Author a, a.LastName l, b.Title t`⟩. The set r contains three elements, one for each branch in s: r = {⟨`Library.Books s, s.Book b`⟩, ⟨`b.Author a, a.LastName l`⟩, ⟨`b.Title t`⟩}. For a graphical depiction see Figure 6.1.

We assume in our algorithms that the Lindex and Bindex operators are supported by the required indexes (recall Chapter 2, Section 2.5.2) for all labels appearing in our path expressions. We do not consider the Pindex, Vindex, or Tindex access methods. The Vindex and Tindex methods are similar to Bindex, and can be used (if we considered full queries) when the appropriate index exists and an appropriate predicate appears in the query's *where* clause. Incorporating these operators into our algorithms is straightforward. Incorporating the Pindex is more complex and is left as future work. We also restrict the join methods considered by the algorithms in this section to nested-loop join (NLJ) and sort-merge join (SMJ). Recall from Chapter 3 that in many cases NLJ is a dependent join that does not contain an explicit join condition and passes bound variables from left to right.

Recall from Chapter 2 (Section 2.4.1) that path expressions in Lorel begin with a *name*, which identifies an entry point and corresponds to a unique object in the database. As explained in Chapter 2 (Section 2.4.7) the query path expression "`Library.Books b`", where `Library` is a name, becomes two path expression components: ⟨`Root`, `Library`, l⟩, ⟨l, `Books`,

$b\rangle$. For the algorithms in this section we combine path expression components such as the above to $\langle$`Library`, `Books`, $b\rangle$. That is, we no longer use the special symbol `Root` and we allow the source variable for a path expression component to contain either a variable or a name. To map such path expressions directly to query plans, we extend the functionality of the physical operators *Scan*, *Bindex*, and *Lindex* to now locate named objects as well as explore the *subpath* for the path expression component: The *Scan* operator must be able to locate a named object and begin searching for descendants from that object. The *Lindex* operator must be able to verify that an ancestor object is a named object. The *Bindex* operator must find all edges in the database with a given label and confirm that the source object for the edge is a given named object.

We could use an exhaustive algorithm to enumerate plans for a given branching path expression: we consider all possible orderings of the components, all possible access methods, and all possible join methods. The total number of left-deep plans is then $n!3^n2^{n-1}$, where n is the number of components, and there are 3 access methods and 2 join methods; creating bushy plans of any type [OL90] increases the search space further. Many of the permutations found in the exhaustive plan space result in plans that are not *valid*, due to incompatible access methods or incorrect use of join operators. Even when we eliminate the invalid plans, the size of the exhaustive plan space is prohibitively large for $n > 5$.

6.2.2 Plan Selection Algorithms

Assuming left-deep query plans only, a plan is characterized by the order of the components, the assignment of an access method to each component, and the assignment of join methods connecting the access methods. An exhaustive algorithm searches the entire space, estimates the cost of each plan, and returns the predicted optimal plan. In this section we present six additional algorithms that heuristically reduce the search space in a variety of ways. The running time for each algorithm is dominated by the size of the plan space that is searched. We present the algorithms roughly in decreasing order of running time, and thus in decreasing amount of plan space explored. However, the search space is pruned in different ways for each algorithm, and the search space for an algorithm is not a subset of the search space for the previous algorithm. We also present four post-optimizations that can be applied to a plan generated by any of our algorithms, although we focus on their effectiveness when applied after two of our six algorithms.

Most of our algorithms generate left-deep plans only, and we are not searching the plan space for alternative plan shapes. The exceptions are Algorithm 2, which may swap left and right subplans in some situations, and Algorithm 5 which, although it searches a relatively small amount of the plan space, can produce some bushy plans.

The algorithms we have designed and the plan spaces they explore were inspired by our observation of queries posed to the Lore system. There are many other ways to reduce the search space and many ways to combine our algorithms. We believe the algorithms and post-optimizations presented here are an interesting representative sample, as confirmed by our experiments presented in Section 6.2.4.

Functions and Classes

Many of our algorithms make use of the following data structures and functions.

- *Bindings* is a data structure that specifies, for each variable, whether the variable is bound or not, and how it was bound (i.e., by which operator).
- The function *OptimalAccessMethod* accepts as input a path expression component, p, and a bindings structure, b. It considers each of the three access methods *Scan*, *Lindex*, and *Bindex* for p, determining whether the access method is valid (based on b) and an estimated cost. *OptimalAccessMethod* returns the valid access method with the lowest cost, and modifies b accordingly.
- The function *OptimalJoin* accepts as input two subplans and produces as output a single subplan that joins the two input plans. The root operator of the result is either *NLJ* or *SMJ*, whichever is valid and estimated to have lower cost.
- *Cost* is a structure containing I/O cost and CPU cost. Comparison operators for cost structures are detailed in Chapter 3 (Section 3.4.3).
- The function *GetCost* accepts as input a single subplan and produces as output the estimated cost of the subplan, as determined by the cost formulas defined in Chapter 3 (Section 3.4.3).

Algorithm 0: Exhaustive

As a measure against which we can compare plans produced by the other algorithms, we consider an exhaustive search of the plan space (Figure 6.2). Recall that the total number

```
function Exhaustive(s)⇒ Plan
        Cost leastCost = COST_MAX;
        Plan bestPlan;
        foreach s′ possible ordering of s do
            foreach assignment a of access methods to components in s′ do
                foreach assignment j of join methods to adjacent components in s′ do
                    Plan current = BuildPlan(s′, a, j);      // Build the actual plan
                    Cost c = GetCost(current);
                    if (c < leastCost)
                        leastCost = c;
                        bestPlan = current;
    return bestPlan;
```

Figure 6.2: Pseudocode for the exhaustive algorithm

of plans considered by the exhaustive algorithm is $n!m^n j^{n-1}$, for n components, m access methods, and j join methods. However, some of these plans are not valid since they violate constraints imposed by the selected access or join methods and the component order (recall Section 6.2.1). Although not shown explicitly, each of our algorithms checks the validity of each plan considered (e.g., within procedure `BuildPlans` in Figure 6.2). Recall that all algorithms take as input a branching path expression expressed as a list s of components.

Algorithm 1: Semi-exhaustive

The motivation for our "semi-exhaustive" algorithm is to continue generating all possible component orderings, but reduce the number of access method permutations. The algorithm considers all possible component orderings and combinations of join methods, but assigns access methods greedily for each ordering and join method permutation. This approach replaces the m^n (access method selection) term in the exhaustive search with 1, resulting in $n!j^{n-1}$ plans considered. The semi-exhaustive algorithm is shown in Figure 6.3. Access method selection is performed in lines 8–11 in Figure 6.3 by performing a single scan of the components, in order, and assigning to each the best access method using *OptimalAccess-Method*.

While a significant portion of the plan space is pruned in the semi-exhaustive algorithm, the running time may still be prohibitively large due to the $n!$ term. Also, the locally optimal access method decisions are not always globally optimal. For example, the cost of a single component in isolation is never lower for *Bindex* than for *Scan* or *Lindex* (when *Scan* or *Lindex* can be used). However, there are situations where a more expensive *Bindex* followed

```
function Semi-exhaustive(s)⇒ Plan
        Cost leastCost = COST_MAX;
        Plan bestPlan;
        int iLength = sizeof(s);
        Operators a[iLength];
        Bindings b, emptyBindings;
        foreach s' possible ordering of s do
            // Choose the best access methods in linear time for this ordering
            b = emptyBindings;
            for(i = 0; i < iLength; i + +)
                a[i] = OptimalAccessMethod(s'[i], b);
            foreach assignment j of join methods to adjacent elements in s' do
                Plan current = BuildPlan(s', a, j);
                Cost c = GetCost(current);
                if (c < leastCost)
                    leastCost = c;
                    bestPlan = current;
  return bestPlan;
```

Figure 6.3: Pseudocode for the semi-exhaustive algorithm

by SMJ with the rest of the plan has lower overall cost than using a *Scan* or *Lindex* as the first access method.

Algorithm 2: Exponential

Algorithm 2 is the algorithm obtained when Lore's original optimizer (Chapter 3) is applied to a path expression. The algorithm reduces the $n!$ term by considering a subset of the possible component orderings. The algorithm generates different component orderings by swapping the order between the first $n-1$ components and the last component, recursively over the input list s. This approach reduces the component ordering term to 2^{n-1}. Figure 6.4 shows precisely how the search space is reduced. Procedure `RecOpt` accepts a list of components and a list of variables currently bound. Two plans are produced. *p1* is the plan where s without its last component is optimized via a recursive call, then joined with the best access method for the last component. *p2* is the converse: an access method for the last component in s is chosen, then joined with the selected plan for the remainder of s. Key to constructing the subplans recursively is the bound variable structure b, which tracks the variables that are currently bound and has a strong influence over the selected access methods for later components. Besides reducing the number of orderings considered, this algorithm also reduces the permutations of join and access methods considered by making

```
function Exponential(s)⇒ Plan
        // Create a structure to track the bound variables, initially empty
        Bindings b;
return RecOpt(s, b);

function RecOpt(s, Bindings b)⇒ Plan
        // If s has a single component then choose the best access method
        int l = lengthof(s);
        if (l==1)
           return OptimalAccessMethod(s[1],b);      // Modifies bindings in b
        // Otherwise, create a plan for the left-then-right order by optimizing s[1..l − 1]
        // and then s[l]
        Bindings b1 = b;
        Plan p1LHS = RecOpt(s[1..l − 1, b1);              // Modifies bindings in b1
        Plan p1RHS = RecOpt(s[l], b1);                    // Modifies bindings in b1
        Plan p1 = OptimalJoin(p1LHS, p1RHS);
        // Create a plan for the right-then-left order by optimizing s[l] then s[1..l − 1]
        Bindings b2 = b;
        Plan p2LHS = RecOpt(s[l], b2);                    // Modifies bindings in b2
        Plan p2RHS = RecOpt(s[1..l − 1, b2);              // Modifies bindings in b2
        Plan p2 = OptimalJoin(p2LHS, p2RHS);
        if (GetCost(p1) < GetCost(p2))
           b = b1;
           return p1;
        else
           b = b2;
           return p2;
```

Figure 6.4: Pseudocode for the exponential algorithm

```
function Polynomial(s)⇒ Plan
        Bindings bEmptyBinding;
        Plan finalPlan;
        while (!empty(s)) do
            Cost leastCost = COST_MAX;
            Component bestComponent;
            Plan bestPlan;
            // Find the component currently in s with the least-cost access method
            foreach e in s do
                // OptimalAccessMethod will modify bTemp, so each iteration must
                // start with an empty binding.
                Bindings bTemp = bEmptyBinding;
                Plan p = OptimalAccessMethod(e,bTemp);
                Cost c = GetCost(p);
                if (c < leastCost)
                    bestComponent = e; bestPlan = p; leastCost = c;
            // Remove the chosen component
            s −= bestComponent;
            // Add the bindings and add the chosen component to the final plan using
            // the best join method
            AddBindings(b, bestComponent);
            finalPlan = OptimalJoin(finalPlan, bestPlan);
    return finalPlan;
```

Figure 6.5: Pseudocode for the polynomial algorithm

locally optimal decisions with respect to a given set of bound variables. Note that when plan $p2$ is chosen over $p1$, then a non-left-deep plan is constructed.

Note that this algorithm is sensitive to the order that the components appear in input list s. The post-optimizations described in Section 6.2.3 specifically address this issue.

Algorithm 3: Polynomial

Our next algorithm reduces the plan space even more aggressively than Algorithms 1 and 2. It combines component order, access method, and join method selection into an $O(n^2)$ operation. The algorithm, shown in Figure 6.5, makes a greedy decision about which component is next and which access and join methods are chosen through each iteration of the `while` loop. The inner `foreach` loop finds the cheapest access method for each remaining component, based on the current bound variables. The component with the least cost is then added to the plan, its variables are marked as bound, and the component is removed from further consideration. For example, given $s = \langle$`Library.Books s, s.Book b, b.Author a, a.LastName l,` $\ldots\rangle$, the component with the least cost access method may be a *Bindex* over `LastName`. In the next iteration variables a and l are bound. At that point a *Lindex* over `Author` might have least cost; if so, b becomes bound, and a join method for variable a is selected.

Obviously, this very greedy approach can produce suboptimal plans in some situations. For example, consider $\langle$`..., b.Author a, a.PhoneNumber p,...`$\rangle$. Suppose there are many `PhoneNumber`'s and `Author`'s in the database, but very few authors have given their phone numbers. The optimal plan may include an *Bindex* for `PhoneNumber` and then a *Lindex* for `Author`, but the polynomial algorithm probably would not consider this plan since the *Lindex* cannot be chosen before the *Bindex* (due to the bound variable restriction), and the *Bindex* is unlikely to be cheapest at any point during the iteration.

Algorithm 4: Bindex-Start

Because the *Bindex* access method requires no bound variables, it is possible to use a *Bindex* to "start" the evaluation of a path expression at any point, then use the *Scan* and *Lindex* access methods to "spread out" and bind the remaining components. The heuristic behind our next algorithm is to first identify those components in s that make good *Bindex* starting points. Let us defer for a moment the definition of "good" starting points and the mechanism by which we choose them. Once we have the *Bindex* starting points, we make a simple linear-time decision for each pair of starting points about whether to use a complete *Scan*-based or complete *Lindex*-based plan between them.

The pseudocode for this algorithm appears in Figure 6.6. The starting points are selected (discussed below) and the chosen components are copied into the set p. The first `foreach` loop in Figure 6.6 considers each adjacent pair of starting points in p, where components e_1 and e_2 in p are considered adjacent if there is a sequence of components in s that leads from the destination variable of e_1 to the source variable of e_2 without using another component in p (i.e., without going through another starting point). For $\langle e_1, e_2 \rangle$ we generate two subplans: the first assigns *Scan* to every component connecting e_1 and e_2, and the second assigns *Lindex* to every connecting component. The best join methods are selected, and the subplan with the lower cost is added to the final plan. Note that if a component is shared by multiple connecting paths then it keeps the first access method selected. Finally, remaining unassigned components are assigned the *Scan* access method in sorted order according to extent size, respecting bound variable restrictions.

Key to the success of this algorithm is identifying those components that make good

```
function Bindex-start(s)⇒ Plan
        Plan finalPlan;
        Set⟨Component⟩ p;
        SortBasedOnSize(s);
        p = ChooseStartingPoints(s);
        // Connect each adjacent pair via all Scan or all Lindex methods (depending on cost).
        foreach adjacent pair ⟨e1, e2⟩ in p do
            Plan p1 = AssignScanandJoin(s,p,e1,e2);
            Plan p2 = AssignLindexandJoin(s,p,e1,e2);
            if (GetCost(p1) < GetCost(p2))
                finalPlan = OptimalJoin(finalPlan, p1);
            else
                finalPlan = OptimalJoin(finalPlan, p2);
        // Assign Scan to remaining components in order of increasing estimated size
        foreach e in s but not in finalPlan do
            Plan temp = AssignScan(e);
            finalPlan = OptimalJoin(finalPlan, temp);
return finalPlan;
```

Figure 6.6: Pseudocode for the Bindex-start algorithm

Bindex starting points. Procedure `ChooseStartingPoints` is shown in Figure 6.7. Recall from Figure 6.6 that when this procedure is called, the components in s have been sorted by the size of their extents. The procedure selects a k, $0 \leq k \leq n$, such that the first k components in s are the starting points. It does so by incrementing k until the ratio between the sizes of the kth and $(k-1)$st extents is below some threshold. That is, we accept the kth component as a good starting point as long as the increase from the size of the previous extent isn't too large. We denote the size of the kth extent as z_k, and set $z_0 = 1$. The procedure is complicated by two details. First, the initial increase from $z_0 = 1$ to a $z_i > 1$ can be very large, so we define a special threshold for this case. Second, if the extents grow at a steady rate below our ratio threshold, then `ChooseStartingPoints` will determine that all components should be assigned the *Bindex* access method. Thus, we set an absolute maximum on starting point extent size based on the first $z_i > 1$.

Again, choosing a good set of *Bindex* starting points is crucial. Note that the constants in Figure 6.7, `INITIAL_CUTOFF`, `RATIO_CUTOFF`, and `TOTAL_CUTOFF` are "tuning knobs", and they required some adjusting before appropriate settings were obtained. However, our current settings result in good performance for a wide variety of database shapes and queries.

The complexity of the Bindex-start algorithm is $O(n \log n)$, and as we will see in Section 6.2.4 it tends to perform well in overall (optimization plus execution) time.

```
function ChooseStartingPoints(s)⇒ Set⟨Component⟩
      int k = 0;
      Boolean first = TRUE;
      int nontrivial;
      for(i = 1; i <lengthof(s); i++)
         if (first)
            if (z_i!=1)
               first = FALSE;
               nontrivial = z_i;
               if (z_i > INITIAL_CUTOFF) break;
         else
            if (z_i / z_{i-1} > RATIO_CUTOFF) break;
            if (z_i > TOTAL_CUTOFF * nontrivial) break;
         k++;
      // Copy the first k components into the result
      Set⟨Component⟩ result;
      for(i = 1; i ≤ k; i + +)
         result.Add(s[i]);
return result;
```

Figure 6.7: `ChooseStartingPoints` used by the Bindex-start algorithm

Algorithm 5: Branches

Our next algorithm optimizes each branch in s in isolation. Optimal subplans for each branch are then combined into a final plan in order of subplan costs, using the cheapest join method between subplans. Pseudocode is shown in Figure 6.8. `Decompose` identifies the individual branches in s, as described in Section 6.2.1. We have chosen our polynomial algorithm (Algorithm 3) to optimize the individual branches, although any of the other algorithms could be used. Note that we are not concerned about one branch relying on bindings passed from another, since each branch is optimized separately. A disadvantage to this approach is an overreliance on the *Bindex* access method, since at least one *Bindex* must appear in the subplan for each branch except the first.

Algorithm 6: Simple

Finally, we consider for comparison purposes a very simple $O(n \log n)$ algorithm that searches only a tiny fraction of the plan space. The algorithm, shown in Figure 6.9, first sorts the components in s by the size of their extents, and this becomes the join order. A single pass through the sorted list assigns the best access and join methods, in a greedy fashion, based on the current bound variables.

```
function Branches(s)⇒ Plan
        Plan finalPlan;
        int numBranches;
        r = Decompose(s, numBranches);
        // One subplan for each branch optimized using Algorithm 3
        Plan subPlan[numBranches];
        int count = 0;
        foreach l in r do
            subPlan[count] = Polynomial(l);
            count++;
        // Sort the array of subplans based on their costs
        SortBasedOnCost(subPlan);
        // Join the subplans together
        for i = 1 to numBranches
            finalPlan = OptimalJoin(finalPlan, subPlan[i]);
return finalPlan;
```

Figure 6.8: Pseudocode for the branches algorithm

```
function Simple(s)⇒ Plan
        Plan finalPlan;
        Bindings b;
        SortBasedOnSize(s);
        // Assign access and join methods in single scan
        foreach e in s do
            Plan tempPlan = OptimalAccessMethod(e,b);      // Modifies bindings in b
            finalPlan = OptimalJoin(finalPlan, tempPlan);
return finalPlan;
```

Figure 6.9: Pseudocode for the simple algorithm

6.2.3 Post-Optimizations

We now introduce four post-optimizations that transform complete plans for path expressions into equivalent plans with the same or lower cost by moving access methods to more advantageous positions within the plan, and reassigning join methods as appropriate. The four post-optimizations are divided into two pairs based on the granularity at which they operate. *Branch post-optimizations* move entire subplans that correspond to complete branches in the original path expression. *Component post-optimizations* move individual access methods. Each optimization technique accepts as input a physical query plan and the original set s of path expression components, producing as output a new physical query plan. In all of the optimizations presented below, when portions of the query plan are reordered new join methods between subplans may be required. New join methods are assigned using *OptimalJoin* (Section 6.2.2), which picks the valid join method with the lowest estimated cost.

Branch Post-optimizations.

Let us assume that we have our set r of branches of s (computed as described in Section 6.2.1), and let l be the size of r, i.e., l is the number of branches in s. Note that the access methods corresponding to the components of a given branch may not be adjacent in the plan we start with, but we can collect the access methods for a branch and place them elsewhere in the plan as long as bound variable restrictions are met. When bound variable restrictions are not met, the corresponding reorderings are not considered.

Post-optimization A. A simple greedy heuristic, running in $O(l^2)$, reorders the branches in the plan. The heuristic estimates the cost of the subplan for each branch in r, and appends to a new final plan the cheapest subplan that does not rely on a branch not yet in the new final plan. This procedure repeats until all branches are in the final plan.

Post-optimization B. This post-optimization is more thorough and therefore more expensive. It constructs and costs all possible reorderings of the branches. There are $O(l!)$ such orderings, but l is usually small in comparison to n (the number of components), and many of the reorderings may be invalid since the subplan for a branch may depend on other branches being executed before it.

Component Post-optimizations.

As with the branch post-optimizations, there are two ways to search the additional plan space.

Post-optimization C. Analogous to post-optimization A but operating at the component level, in $O(n^2)$ time we repeatedly find the component with the smallest cost that does not rely on a component not yet in the new final plan, and append the access method associated with that component to the new final plan. The process repeats, with new cost estimates for the remaining components, until all components have been placed.

Post-optimization D. Analogous to post-optimization B but operating at the component level, all possible valid reorderings of the components are considered. In general this can add an additional $n!$ to the running time, but in practice, since access methods have already been assigned to the components, the number of valid reorderings is limited.

We will evaluate the effectiveness of these post-optimizations when applied to plans generated by Algorithms 2 and 3. Algorithm 2 (the exponential algorithm) can benefit greatly from these post-optimizations, because the quality of the initial plan produced is sensitive to the order of the components in input s. Since Algorithm 3 combines component order and access method selection into a single pass, the post-optimizations provide a "second chance" to reorder the components without also deciding the best access methods.

6.2.4 Experimental Results

We implemented the six algorithms and four post-optimizations presented in Sections 6.2.2 and 6.2.3, using the Lore infrastructure but separate from the Lore optimizer described in Chapter 3. We performed a variety of experiments over data and path expressions of varying shapes. We report on the times required to construct query plans along with query execution times.

Setting

We use the synthetic Movie Store database introduced in Chapter 2 (Section 2.3) and shown in Figure 2.4. We provide more details here about the structure of the database to ensure understanding of our experiments. There are over 12,000 movies in the database. Each movie has as subobjects people who acted in the movie, locations where the movie was shot, and stores where the movie is available for rent. Each of the 256 store objects has as subobjects store location and the company that owns the store. There are only 13 companies that own stores, although the database contains more than 150 companies (companies that don't own stores are related to the movie industry in other ways). Companies contain as subobjects the people who work for that company. Each person has a subtree containing personal information, including things that they like and dislike.

The shape of the data is very important. It is highly graph-structured, with a unique entry point named `MovieStore`. There is a very small first-level fan-out to distinguish between different categories in the data (e.g., all movies in the database are reachable via "`MovieStore.Movies`", and all companies are reachable via "`MovieStore.Companies`"). The data then fans out rapidly since there are thousands of movies, hundreds of companies, thousands of people, etc. The data then gets even wider or narrows substantially, depending on the path taken. For example, the data narrows when we look for all the stores that rent movies because there are only 256 of them, although the number of "`MovieStore.Movies.Movie.AvailableAt`" paths is huge. The data narrows even further if we consider "`MovieStore.Movies.Movie.AvailableAt.OwnedBy`", since franchises own many stores. However, the data fans out again if we explore the franchise employees via the path "`MovieStore.Movies.Movie.AvailableAt.OwnedBy.Employee`". Our experience is that this "narrow-wide-narrow" pattern appears commonly in graph-structured data.

All experiments were conducted using Lore on an Intel Pentium II 333 mhz machine running Linux. The database size was 12 megabytes, and the buffer size was set to 40 pages of 8K each, or about 2% of the size of the database.

Overall Results

We ran each algorithm except the exhaustive one, including Algorithms 2 and 3 augmented with Post-optimizations A–D (denoted 2A, 2B, etc.), on the sample set of 8 branching path expressions shown in Figure 6.10. For each of the 8 experiments, we ranked the algorithms based on the time to execute the chosen plan, and also the total time to both select and execute the plan. We then added together the ranks for each algorithm across all 8 experiments, treating each query as equally important. The results are shown in Table 6.1.

Algorithm 4, the Bindex-start algorithm (marked by ** in Table 6.1), performs the best. In terms of plan execution speed it ranks second, just behind Algorithm 2D (marked by *). Algorithm 4 ranks first for total time, which includes the time required for optimization.

1. MovieStore.Movies s, s.Movie m, m.Actor a, m.AvailableAt t
2. MovieStore.People s, s.Person p, p.Name n, p.Phone z, p.Likes l, l.Thing t
3. MovieStore.People s, s.Person p, p.Likes l, l.Thing t2, p.Dislikes d, d.Thing t1
4. MovieStore.Stores x, x.Store s, s.Name n, s.Location l, l.City c
5. MovieStore.Movies s, s.Movie m, m.Sequel s, s.AvailableAt a, a.OwnedBy o, o.Affiliated f, f.Phone p
6. MovieStore.Movies s, s.Movie m, m.Sequel s, s.AvailableAt a, a.OwnedBy o, o.Affiliated f, f.Name n
7. MovieStore.Movies s, s.Movie m, m.Actor a, a.Likes l, l.Thing t, a.Address d, m.Title z
8. MovieStore.Companies s, s.Company c, c.Affiliated a, c.Name n

Figure 6.10: Sample set of 8 branching path expressions

Note that Algorithm 2D is ranked eleventh in total time: Algorithm 2 (the exponential algorithm) explores a fairly large portion of the search space, and Post-optimization D is the most expensive post-optimization. (Further experimental results for Post-optimization D are reported later.)

In two experiments Algorithm 4 created the fastest plan, but in other instances it ranked in the top three or four. Its strength is that it consistently selected good plans in a reasonable amount of time. Overall the plans produced by Algorithms 5 and 6 (the branches and simple algorithms) performed poorly, as shown in the last two rows of Table 6.1. Although both algorithms did produce very good plans for a small number of queries, the results were inconsistent. Unfortunately, we have not been able to characterize the situations in which these algorithms perform well—it appears to depend on complex interactions between query shape and detailed statistics about the data.

Another interesting result from Table 6.1 is the poor overall performance of Algorithm 2, the exponential algorithm, without post-optimizations. Recall that Algorithm 2 reduces the component orderings considered from $n!$ to 2^{n-1}.

The high overall times for Algorithm 1 were expected since optimization time is prohibitively large. However, the slow plans produced by Algorithm 1 were unexpected. Apparently making a local access method decision for a given component order ignores the global situation too often.

Note the anomaly in the results of Table 6.1 for the execution times of Algorithms 2A and 2B, reported as 9th and 10th respectively. Since 2B explores a strictly larger plan

Algorithm	Execution Time Rank	Total Time Rank
1	11	14
2	14	13
2A	9	9
2B	10	11
2C	3	5
2D *	1	11
3	6	2
3A	8	4
3B	7	6
3C	5	2
3D	3	9
4 **	2	1
5	13	12
6	7	8

Table 6.1: Overall results

space than 2A we would expect it to produce strictly better plans. We attribute this slight inconsistency to somewhat imperfect statistics and/or cost estimates.

More Detailed Results

In this section we look in more detail at some experiments focusing on Algorithms 1–6 without considering post-optimizations.

Experiment 6.2.1 (Simple Branching Path) In our first experiment $s = \langle$`MovieStore.Movies s, s.Movie m, m.Actor a, m.AvailableAt v`$\rangle$. This expression contains three branches. In our database, on average there are more actors that acted in a movie than stores that carry that movie. Thus, it is usually beneficial for a plan to evaluate the branch "`m.AvailableAt v`" before "`m.Actor a`" (to keep intermediate results smaller). Table 6.2 shows the optimization, execution, and total time (in seconds) for each of the algorithms, ranked by total time.

Algorithm 5, the branches algorithm, generates the best plan and does so quickly. This plan uses *Bindex* for `AvailableAt`, then SMJ with a *Scan*-based plan for `MovieStore.Movies.Movie`. A final SMJ with a *Bindex* for `Actor` completes the plan. This plan performs well in this particular case because most of the data discovered by each branch

Rank	Algorithm	Optimization time	Execution Time	Total Time
1	5	0.445	41.770	42.215
2	3	0.099	48.573	48.672
3	4	0.145	48.573	48.718
4	1	1.180	48.573	49.753
5	2	0.108	60.643	60.751
6	6	0.318	108.600	108.918

Table 6.2: Results for Experiment 6.2.1

Rank	Algorithm	Optimization time	Execution Time	Total Time
1	6	0.0741	0.0729	0.147
2	4	0.104	0.127	0.231
3	3	0.1108	0.136	0.247
4	5	0.085	1.241	1.326
5	2	0.26	1.996	2.256
6	1	174.749	1.38	176.129

Table 6.3: Results for Experiment 6.2.2

independently actually contributes to the final result. Thus, optimizing branches independently does not cause significant irrelevant portions of the database to be explored. Algorithm 6, the simple algorithm, does very poorly. It first selects *Bindex* for `AvailableAt`, then *Lindex* for `Movie` and `Movies`, then *Scan* for `Actor`. The better plans verify that an object has both `AvailableAt` and `Actor` subobjects before working backwards to match `MovieStore.Movies.Movie`. Algorithms 1, 3, and 4 all produced the same plan for this experiment, so here and in subsequent results where the plans were the same, we averaged their slightly deviating execution times. □

Experiment 6.2.2 (More Branches) In our second experiment s = ⟨`MovieStore.People s, s.Person p, p.Name n, p.Phone h, p.Likes l, l.Thing t`⟩. In our database each person has a single name, and roughly half of the people have things that they like. On average, those with likes have four of them. Most people in the database do not have a phone number. The results of this experiment are shown in Table 6.3.

Algorithm 6 happened to do well in this case, in contrast to the first experiment where it had the worst execution time. It first chose *Bindex* for `Phone` (because there aren't many

Rank	Algorithm	Optimization Time	Execution Time	Total Time
1	6	0.07	2.117	2.1875
2	4	0.085	6.932	7.017
3	2	0.264	6.932	7.196
4	5	0.143	7.098	7.241
5	3	0.096	19.551	19.647
6	1	161.274	5.354	166.628

Table 6.4: Results from Experiment 6.2.3

in the database), then *Scan* for `Likes` which immediately narrows the search to people that have both a phone number and some likes. Other algorithms did not find this plan for various reasons. Algorithm 4, the Bindex-start algorithm, also did well. It chose `People` and `Phone` as starting points with a *Lindex*-based plan between them, and *Scan*'s for `Name` and `Likes`. □

Experiment 6.2.3 (Longer Branches) In our third experiment s = ⟨`MovieStore.People s, s.Person p, p.Likes l, l.Thing t1, p.Dislikes d, d.Thing t2`⟩. Most people in the database have either likes or dislikes, but few have both, so this is a situation in which treating branches as indivisible units results in poor plans. Results are shown in Table 6.4.

Algorithm 6 again produces a good plan (the same plan is produced by Algorithm 3C, not shown in the table). In this plan, a *Bindex* for `Dislikes` followed by a *Scan* for `Likes` narrows the search to people that have both likes and dislikes, without discovering yet the actual things that they like/dislike. It is the interleaving of the execution of branches in the plan that results in good execution times. Poor decisions are made by Algorithms 2 and 3, which choose *Scan*-based plans. Algorithm 5 does poorly because it requires branches to be executed indivisibly. □

Experiment 6.2.4 (Weakness of the Bindex) Our fourth experiment illustrates the weakness inherent in overusing the *Bindex* access method. While several *Bindex* operators joined using SMJ's can be competitive against multiple *Scan* operators with NLJ's, a major drawback is that *Bindex* always considers all occurrences of a given label. Consider s = ⟨`MovieStore.Stores x, x.Store s, s.Name n, s.Location l, l.City c`⟩. A *Bindex* for `Location` fetches not only the locations for stores, but also locations where movies

Rank	Algorithm	Optimization Time	Execution Time	Total Time
1	3	0.071	0.312	0.389
2	2	0.144	0.312	0.456
6	5	0.111	7.122	7.232

Table 6.5: Results for Experiment 6.2.4

were filmed. By contrast a *Scan* for `Location` using bindings for stores does well, since the number of stores in comparison to the number of locations in the database is small. Table 6.5 presents a few results for this experiment.

The best plan in this situation happens to be one with all *Scan* access methods, and all of the algorithms except Algorithm 5 generate this plan. Since Algorithm 5 must optimize each branch separately, it is forced to use *Bindex* for `Location`. Notice that the query shape is actually very similar to Experiment 6.2.1, where Algorithm 5 produced the optimal plan, but the shape and distribution of the data being accessed is very different. □

Post-Optimizations

In general, the post-optimizations improve query execution time at the expense of increased optimization time. As we saw in Table 6.1 with the good performance of the plans produced by Algorithm 2D, the net effect can be a win.

Recall that Post-optimization D is the most thorough, since it operates at the component granularity and doesn't apply any heuristics in its search. It is also the most expensive: it can add a second or even more to the optimization time. In our experiments, it decreased query execution time by an average of 22%, ranging from 0% faster (no change to the plan) to 88.5% faster. Obviously the benefit of post-optimization thus depends on whether the query itself is expected to be expensive.

To be more concrete, let us consider as an example the impact of each of our four post-optimizations on the plan produced by Algorithm 2 for Experiment 6.2.2. Results are shown in Table 6.6. Algorithm 2 without post-optimization does very poorly in this experiment, and after applying Post-optimization D the new plan is almost an order of magnitude faster. However, the trade-off between better query performance and longer optimization time is evident with an increase in total time after post-optimization. In this situation, and in many others, we found that Post-optimizations B and C produce tangible improvements at

Post-optimization	Optimization Time	Execution Time	Total Time
None	0.26	1.996	2.256
A	0.342	0.623	0.965
B	0.364	0.62	0.984
C	0.311	0.24	0.551
D	2.383	0.229	2.612

Table 6.6: Post-optimizations for Algorithm 2 on Experiment 6.2.2

a reasonable cost.

Comparison Against Exhaustive Search

We implemented the exhaustive search strategy described in Section 6.2.2 in order to compare the true lowest (predicted) cost plan against plans chosen by our six algorithms. Since exhaustive search is so expensive, we were limited to considering path expressions with fewer than 6 components, and even 5-component expressions were very slow to optimize. Overall our algorithms produced plans that were competitive with the optimal plan. We ran four representative experiments and calculated how much slower each plan was when compared to the plan selected by the exhaustive algorithm. Table 6.7 shows the average multiplicative increase in query execution time over all experiments when compared with the optimal plan.

We also considered some extreme points. For simple linear path expressions our algorithms did very well. In one case, all of our algorithms except Algorithms 4 and 6 produced the same plan as the exhaustive algorithm, and Algorithms 4 and 6 produced plans that were only 1.05 times slower. In another experiment, none of the algorithms generated the same plan as the exhaustive algorithm, some of the plans were 2 to 3 times slower than the optimal, and Algorithm 5 produced a plan that was nearly 6 times slower. However, as can be seen in Table 6.7, overall our algorithms do produce competitive plans. Furthermore, they do so in a small fraction of the optimization time.

6.3 Improving Path Expression Evaluation Using Groupings

We now consider our second optimization technique designed specifically for path expressions. Recall from Chapter 2 (Section 2.5.1) that physical query operators in Lore operate

Algorithm	Average Times Optimal
1	1.23
2A	1.35
2B	1.12
2C	2.38
2D	1.08
3A	2.19
3B	1.26
3C	2.30
3D	1.29
4	2.05
5	2.25
6	2.60

Table 6.7: Summary of the average times worse than optimal

over *evaluations*, which represent paths through the data currently being explored. We introduce an optimization technique called *grouping introduction* (GI). This technique transforms a physical query plan for a path expression into an equivalent plan with a smaller cost by introducing one or more *Group* operators. The *Group* operator creates an *encapsulated evaluation set* (EES) (recall Chapter 3, Section 3.4.2) from a set of evaluations. Recall that an EES is a set of evaluations where the evaluations are "grouped" according to the objects assigned to one variable (the *primary variable*). For each binding for the primary variable the EES contains a structure that lists the evaluations for other variables (the *secondary variables*).

Although there is some overhead associated with introducing a *Group* operator, overall query execution time can decrease because of savings in subsequent physical operators that operate over the EES instead of over the original evaluations. When the original evaluations are needed, a *ForEach* operator (recall Chapter 3, Section 3.4.2) decomposes the structure containing the secondary variables in the EES. The *Group* operator is similar to the *CreateTemp* operator except the EES created by the *Group* does not necessarily have to be stored on disk. Also recall (from Chapter 3, Section 3.4.2) that the *CreateTemp* operator further encapsulates the EES into a single evaluation.

The GI optimization technique differs from the optimization algorithms in Section 6.2 in that GI is a post-optimization technique applied once a physical query plan has been generated. The plan could have been generated by any of the optimization algorithms presented in Section 6.2 or by techniques described in previous chapters. Although the GI technique can be applied to any physical query plan, it is only effective when a *Group* operator is introduced to create an EES where the primary variable and set of secondary variables are variables bound by a path expression. A comparison of the *Group* operator versus the more broadly applicable *Cache* operator (and more generally the GI optimization versus the subplan caching optimization) appears in Section 6.3.2.

We begin this section by describing in more detail the motivation for this query optimization technique. We then describe the function and placement of the new *Group* physical operator. We conclude with a performance analysis.

6.3.1 Motivation

During execution of a path expression a variable x may be bound to the same object many times. This repetition can occur as an artifact of the particular query plan selected, or it may be inherent in the path expression and shape of the database. Rebinding a variable to the same object may result in duplicate work being done when other variables are bound to objects based on the binding of x. The following two examples, which use the Movie Store database, illustrate this situation.

Example 6.3.1 Recall from Section 6.2.4 that the data in the Movie Store database narrows from a few thousand objects down to a few hundred objects when going from movie objects to store objects. Consider the following path expression.

```
MovieStore.Movies x, x.Movie m, m.Title t, m.AvailableAt s,
     s.Location l
```

Variable s will be bound to each of the movie stores in the database many times since a store can be reached once for every movie that it carries. If we use a *Scan* operator to bind l from s, then we will rediscover the location of each store many times. More generally, given that there are relatively few distinct objects that can be bound to s, but many paths to those objects, a lot of time may be wasted refetching objects bound after s. We call the variable s a *funnel variable* since many paths funnel through a small set of objects.

Note that a bottom-up execution strategy for the path expression above also creates a funnel variable. In this case variable m will be bound to the same movie many times.

Thus, a funnel variable is created as a result of both query execution strategy and database shape. Given a database and path expression, one execution strategy may result in one funnel variable, whereas another strategy may result in a different funnel variable. □

Example 6.3.2 Consider the following path expression, which explores a tree-structured subset of the Movie Store database.

```
MovieStore.Movies s, s.Movie m, m.FilmLocation f, f.City, f.State
```

Suppose that the optimal plan first finds all `FilmLocation` edges in the database (via the *Bindex* operator) and then binds the subpath "`MovieStore.Movies s, s.Movie m`" in reverse order. Once the path to the named object `MovieStore` is discovered then the city and state can be fetched. This execution strategy results in funnel variable m because a movie may have had many different location shoots, resulting in m being bound to the same object many times. In this case the reverse evaluation of the subpath "`MovieStore.Movies s, s.Movie m`" will occur many times for each movie. □

Our solution to improving the performance in the presence of funnel variables involves creating an EES (recall Chapter 3, Section 3.4.2) with the funnel variable serving as the primary variable. As an example, consider the database fragment in Figure 6.11(a). The tree in Figure 6.11(a) is a subset of the Movie Store database, shown in more detail. Suppose we are executing a query plan that uses a *Bindex* operator for the `State` edges. A *Bindex* would discover that objects &9, &12, &15, &18 and &21, in Figure 6.11, have incoming edges labeled State. Using *Lindex* operators to bind up to variable m results in the five evaluations shown in Figure 6.11(b). Creating an EES with variable m as the primary variable and f as the only secondary variable results in only two evaluations as shown in Figure 6.11(c). Further query execution using the EES only needs to match the path above each distinct m a single time. When objects bound to f are required by the plan then a *ForEach* operator must be introduced.

6.3.2 Comparison of Grouping Introduction and Subplan Caching

We introduced the subplan caching optimization technique in Chapter 5. Although there are some similarities between the two techniques, GI and subplan caching are fundamentally

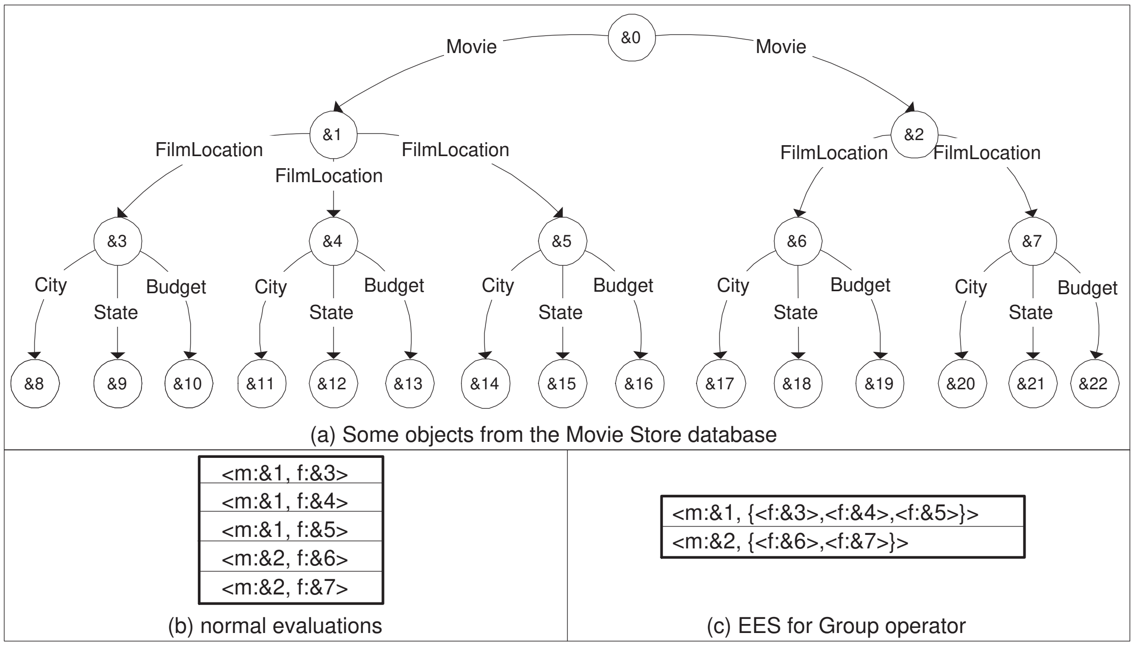

Figure 6.11: Some objects from the Movie Store database

GI	Subplan Caching
Sorts evaluations from subplan in order to create an EES	Uses a hash table to store cache
May require multiple sort runs	May remove cache entries and require reexecution of subplan
Evaluations in EES affect all subsequent operators up to *ForEach* operator	Effect of cache is localized to subplan
Applicability of GI limited, but when it can be applied it is usually more beneficial	*Cache* improves performance in a wide range of situations

Table 6.8: Comparison of GI and Subplan Caching

different. Both techniques cannot be applied at the same point in a query plan simultaneously, although one technique can be applied in a subplan of the other. There are some situations where either technique can be applied, and each has particular strengths. Table 6.8 summarizes the differences between the two optimization techniques. We discuss each row in Table 6.8 in more detail.

- The *Group* operator is sorting-based while the *Cache* operator is hash-based. The issue of sorting versus hashing is well-studied, e.g., [Gra93].

- Subplan caching uses a fixed-size hash table, thus victim selection and the effects of removing the victim elements must be considered. Grouping introduction may require additional disk I/O when the EES is larger than the memory allocated for it, but doesn't need to redo work at a later time: a *Group* operator ensures that execution for a given binding to the funnel variable will only occur a single time. By contrast, the *Cache* operator cannot ensure that its subplan will be executed a single time for each binding since it has a finite cache size.

- Recall from Chapter 5 (Section 5.5) that the *Cache* operator reduces the number of evaluations of the subplan of the *Cache* operator. In contrast, the *Group* operator reduces the number of executions for those operators that are executed after the *Group* operator, since the *Group* creates unique bindings to the funnel variable after it has been executed.

- The most important distinction between GI and subplan caching is when the techniques can be applied. Subplan caching is a more general technique—it can be used in conjunction with single access methods, complete clauses of a query, and subqueries. The GI technique is more limited, and is applicable only in the context of path expressions. Proper placement of the *Group* operator requires finding a path expression with a funnel variable that is bound in the *Group* operator's subplan, s, and s must not depend on variables bound earlier. To understand the last requirement, suppose that s does depend on variables bound earlier. As an example, consider the following clause:

  ```
  where x > z.A.B.C
  ```

  Let us focus on path expression "`z.A.B.C`" and assume a top-down execution strategy for this path expression. A subplan cache directly over the subplan for "`z.A.B.C`" can

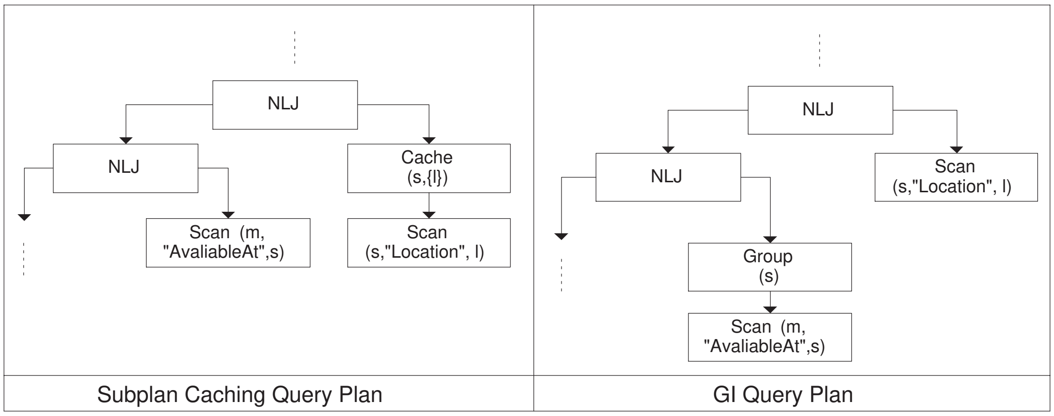

Figure 6.12: Physical query plan segments for both Caching and Grouping plans

cache the set of `C`'s for a given z. This is especially useful when x has changed but z has not since the cache entry for z will not have been removed (recall Chapter 5). Consider the same scenario with a *Group* operator. A *Group* can be placed above the subplan for "`z.A.B.C`". When *Group* is asked for an evaluation it will execute its subplan to exhaustion and group all evaluations based on their binding to z. However, there will only be a single evaluation in the EES since z was bound previously. On the next call to the *Group* operator the same procedure will repeat, with the *Group* operator executing its subplan to exhaustion even if z is bound to the same object as before. Thus, the *Group* operator provides no benefit and only introduces additional I/O and CPU cost in this situation.

There are some situations where either technique can be applied. Consider the path expression in Example 6.3.1, introduced earlier. Suppose that a top-down query execution strategy is used and that subplan caching introduced a *Cache* operator directly above the evaluation of "`s.Location l`". The *Cache* will remember the locations for a store to avoid rediscovering the information when the same store is seen more than once. With a similar effect, GI may introduce a *Group* directly above the binding of variable s. Portions of these two plans are shown in Figure 6.12.

Note that a single *Group* can be used where many *Cache* operators may be required. For example, if the query in Example 6.3.1 had the additional path expression component `s.Name n` in its *from* clause, then the subplan caching query plan in Figure 6.12 would probably introduce another *Cache* operator above the *Scan* for `s.Name n`. The GI query plan would not need to introduce another *Group* operator since the EES would also benefit the *Scan* for `s.Name n`. This difference can be significant because of the overhead associated with each *Cache* operator.

See Section 6.3.5 for a preliminary quantitative comparison of GI and subplan caching.

6.3.3 The Group Physical Operator

The *Group* physical operator produces an EES from a single subplan. Recall from Chapter 3 (Section 3.4.2) that an EES reduces a set of evaluations of the form $\langle v_1 : o_1, v_2 : o_2, ..., v_n : o_n \rangle$ into a set of evaluations of the form $\langle v_1 : o_1, v_g : \{\langle v_2 : o_2, ..., v_n : o_n \rangle\}\rangle$. Recall, also from Chapter 3 (Section 3.4.2), that the *ForEach* physical operator flattens the structure in v_g and creates a set of evaluations equivalent to the original stream of evaluations fed to the operator that created the EES. The *Group* operator can be placed over any subplan s and is characterized by the primary variable and secondary variables for the EES, and by the amount of main memory allocated to perform the grouping.

The basic operation of the *Group* operator is simple. A request is made for the *Group*'s next evaluation by the parent operator of the *Group*, which provides some bindings. The *Group* operator asks s for resulting evaluations until there are no more. The evaluations returned by s are sorted based on the oids of the objects bound to the primary variable. Once sorting is done a complete pass is performed over all sorted evaluations to create the EES. Then the evaluations in the EES are returned one at a time to the *Group*'s parent operator. The *Group* operator introduces additional I/O only when the amount of memory required by the sort operation exceeds the memory allocated to the grouping. A standard multi-pass sort [Gra93] is used in these situations.

6.3.4 Placement of the Group Operator

Recall from Chapter 3 (Section 3.4.1) that each logical query plan node in the Lore system is responsible for creating estimated optimal query plans for its subplan given a set of bound variables. A *Group* operator could be placed above any subplan, however adding a *Group* operator always increases the CPU cost for the subplan. In situations where the number of evaluations expected from the *Group* operator is large then the *Group* can also introduce additional I/O cost due to the multi-pass sort. Overall, the *Group* operator may decrease the cost of the entire query plan, but the savings occur later in the plan than where the

Group operator is placed. Thus, the *Group* operator does not fit well into Lore's model of creating physical query plans since locally optimal decisions are used (recall Chapter 3).

One solution is to adopt the general idea of *interesting orderings* first introduced in [SAC+79]. We could generate n different physical query plans for a logical query plan node, where n is the number of possible "interesting groupings" that could be made. In the context of Lore's query enumeration strategy it would be necessary to augment a logical query plan node to enable the creation of an optimal subplan given that a specific variable must be a group variable at the end of the execution of the subplan. The obvious drawback to this approach is the vast increase in search space size.

Another solution, and the one we adopt, is to heuristically place *Group* operators after the entire optimal physical query plan has been generated. In this solution a post-optimization step heuristically introduces 0 or more *Group* operators to decrease the cost of the entire query plan. The post-processing step to introduce *Group* operators proceeds as follows. First, variables in the physical query plan are assigned a numeric value, f, indicating to what degree they act as a funnel variable. The formula for determining f is discussed in more detail below, but the smaller the f value, the more likely that the variable will act as a funnel variable and therefore benefit from a *Group* operator. The variables in the query are sorted by their f values in increasing order. The first k of these variables are chosen (the actual value used for k is discussed below) and *Group* operators are placed above the binding of each variable. We have not considered issues related to creating an EES inside of another EES, or even creating multiple EES's. Thus, we always use a *ForEach* operator to unnest the secondary variables before another *Group* operator.

This heuristic placement of *Group* operators hinges on the accurate assignment of f to each variable. The f value for a variable x is determined by the following three functions. These functions use the statistics created by Lore that were introduced in Chapter 3 (Section 3.4.3).

1. The estimated distinct number of objects that will be bound to x. This value, $Distinct(x)$, is either $|PathOf(x)|^d$ or $|PathOf(x)|_d$, and the choice depends on the physical operators chosen for the path expression components bound before x that either feed x or are fed by x. If x is bound due to a sequence of *Scan* operators for previous path expression components then $Distinct(x)$ will return $|PathOf(x)|_d$. If x is bound due to a sequence of *Lindex* operators for previous path expression components, then $Distinct(x)$ will return $|PathOf(x)|^d$. If x is fed by a single *Bindex*(l,

v_1, v_2) operator then *Distinct*(x) will return $|PathOf(x)|^d$ if $x = v_1$ and $|PathOf(x)|_d$ when $x = v_2$. In all other cases, we choose one of the two values arbitrarily, although some heuristics might be applied.

2. The estimated total times an object will be bound to x, *Count*(x). This value is the number of paths that reach x given the path expression components bound before x and is computed using $|PathOf(x)|$.

3. The number of variables that are bound as a result of the object bound to x, *Feeds*(x). This value is straightforward to determine from the physical query plan.

We never choose a variable x to be a group variable when *Feeds*(x) is 0. The EES will have no benefit for this variable since the primary variable does not feed another access method. Essentially this means that we would create an EES without benefiting from the unique binding for the primary variable.

Otherwise, f for a variable x is determined by the formula (*Distinct*(x)/*Count*(x)) / *Feeds*(x). Consider the first term: *Distinct*(x)/*Count*(x). The smaller this expression is the more repetition there is in objects bound to x. Further dividing by *Feeds*(x) means that the more path expression components fed by x, the smaller the f value.

Once an f value has been assigned to all variables, the first k in the sorted list are eligible to become group variables. Each variable chosen can result in a *Group* operator. The three cases in which we would not add a *Group* operator since immediate "ungrouping" is required are:

1. A variable y is not chosen if y requires the ungrouping of another group variable x before x is used as a source variable in any access method.

2. A grouping is never done over a variable that will be used immediately in a hash join.

3. A grouping is never performed when one of the partition variables will be used immediately after the grouping.

Finally, we must choose a value for k. It is possible for a complicated query with many path expression components to benefit from a large k, however in the queries we have considered in our experiments, they have only benefited from a single *Group* operator. The optimizer could choose the value of k by examining the distribution of f values. In our current implementation we simply set $k = 1$.

The *ForEach* operator, required after a *Group* operator, should be placed at the latest possible position that still ensures validity of the query plan. This decision can be made based on the usage of variables in the query plan.

6.3.5 Experimental Results

We have implemented the GI technique in the Lore system, with a switch that allows us to optimize a query with and without the GI technique. We call the plan created when GI is active the *GI plan* and the plan created by the Lore optimizer without GI the *normal plan*. We have restricted our experiments to sets of path expressions and not general queries.

The implementation for the *Group* operator follows closely the description given in Section 6.3.3. The heuristic placement of *Group* operators increased query optimization time by less than 5% for most queries.

We tested the GI technique over the Database Group database, introduced in Chapter 2 (Section 2.3) and shown in Figure 2.2. Unless otherwise stated we restricted the size allocated to each *Group* operator to 40K. In several experiments the restricted memory size resulted in multiple sort runs to do the grouping. In all of the experiments below we first optimized and executed the query without enabling the GI optimization to produce a query plan without any *Group* operators. We then executed the same query with GI enabled. We show a graphical representation of the query (similar to the representation introduced in Chapter 4, Section 6.2) for each experiment, augmented to indicate the position of any *Group* operators.

Experiment 6.3.1 We used the following list of path expression components for this experiment:

```
<Root.DBGroup d, d.Member m1, d.Member m2, m1.Project p, p.Member m3>
```

The resulting GI plan for this experiment is shown in Figure 6.13. For this query plan and database d was chosen as the grouping variable. The optimizer placed the *Group* operator directly after d was bound the second time by the *Lindex* operator. We explain in more detail why d was chosen. Table 6.9 contains the set of path expression components along with their corresponding *Count*, *Distinct*, and *Feeds* values for the variable bound by an access method. For the query execution strategy chosen there are only two variables that exhibit any of the characteristics of funnel variables: $m1$ and d. Notice that a grouping of $m1$ has a low f value primarily because it feeds two path expression components, where

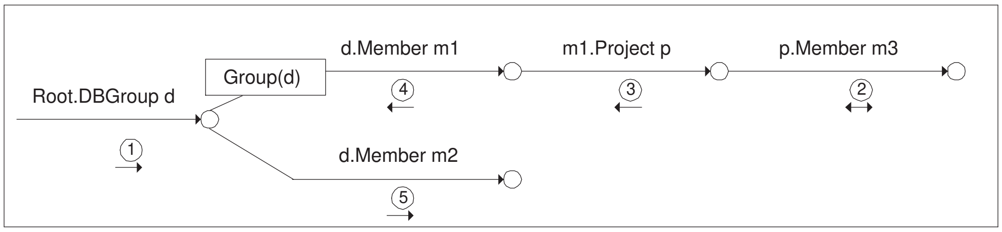

Figure 6.13: Query plan produced for Experiment 6.3.1

PE Component	Access Method	Bound Variable	*Distinct*	*Count*	*Feeds*	*f*
`Root.DBGroup d`	Scan	d	1	1	1	1
`p.Member m3`	Bindex	$m3$	7	50	0	-
`p.Member m3`	Bindex	p	42	50	3	0.28
`m1.Project p`	Lindex	$m1$	441	4953	2	0.04451
*`d.Member m1`	Lindex	d	1	5283	1	0.00018
`d.Member m2`	Scan	$m2$	514	514	0	-

Table 6.9: Statistics for determining the funnel variable for Experiment 6.3.1

d, which appears much later in the execution of the path expression, only feeds one other variable. The clear favor is for d since there is only a single object that will be bound to d (since d is bound to the named object `DBGroup`). This results in a grouping with a single group which tremendously reduces the number of tuples that will be fed to the final access method `Scan(d.Member m2)`. Effectively, we are executing the chosen optimal plan for "`Root.DBGroup d, d.Member m1, m1.Project p, p.Member m3`" completely, deferring execution of "`d.Member m2`" until we have created a grouping for the set of results. This grouping contains only a single evaluation since every named object is unique. The time required to execute the normal query plan, which is exactly like Figure 6.13 without the *Group* operator, was 282.7 seconds. The GI query plan executed in less than half this time at 125.98 seconds. □

Experiment 6.3.2 This experiment is similar to Experiment 6.3.1, however a larger list of path expression components is used.

```
<Root.DBGroup d, d.Member m1, d.Member m2, m1.Name n1, m2.Name n2,
  m1.Project p, p.Member m3>
```

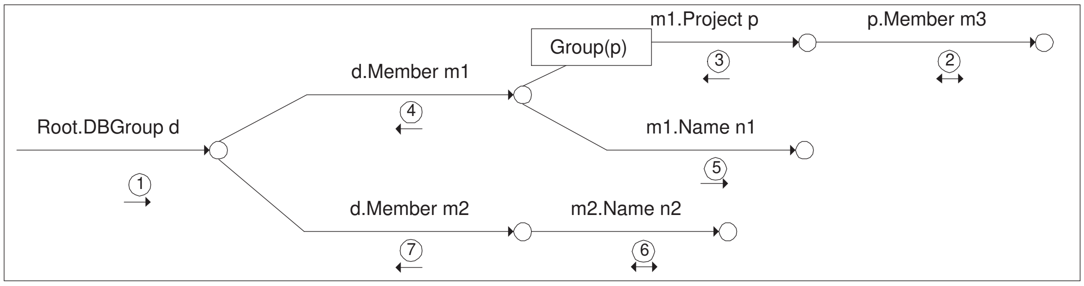

Figure 6.14: Query plan produced for Experiment 6.3.2

The resulting GI plan for this experiment is shown in Figure 6.14. Note that this query plan is different than the query plan for Experiment 6.3.1 and thus the funnel variable is also different. For this plan, variable d is not chosen as a funnel variable since it does not feed any other path expression components. Instead, the *Group* operator is applied over $m1$, since a member can work on many projects and thus can be reached via many paths, and because $m1$ feeds two path expression components: the *Lindex* for `d.Member m1` and the *Scan* for `m1.Name n1`. This *Group* operator produced 441 different groups from a total of 5,339 evaluations, and thus reduced the number of tuples passed on to later query operators by over 90%. The GI plan took 154.41 seconds and the normal plan took an order of magnitude longer at 1730.72 seconds. □

Experiment 6.3.3 Our final experiment gauges the performance difference between the GI and subplan caching techniques. We ran the following list of path expression components using both techniques:

```
<Root.DBGroup d, d.Member m, m.Project p, p.Member m2, m2.Favorites f,
  f.Book b>
```

The query plans produced by both techniques are shown in Figure 6.15. A top-down plan was used in both cases. Subplan caching placed three *Cache* operators, one each above the access methods for `p.Member m2`, `m2.Favorites f`, and `f.Book b`. The GI plan chose p as the group variable since there are many paths to projects, but only a small set of distinct projects. p was also chosen because it feeds three other path expression components. The normal query plan executed in 26.10 seconds; the GI plan executed in 0.40 seconds; The subplan caching plan executed in 15.82 seconds. The results for this

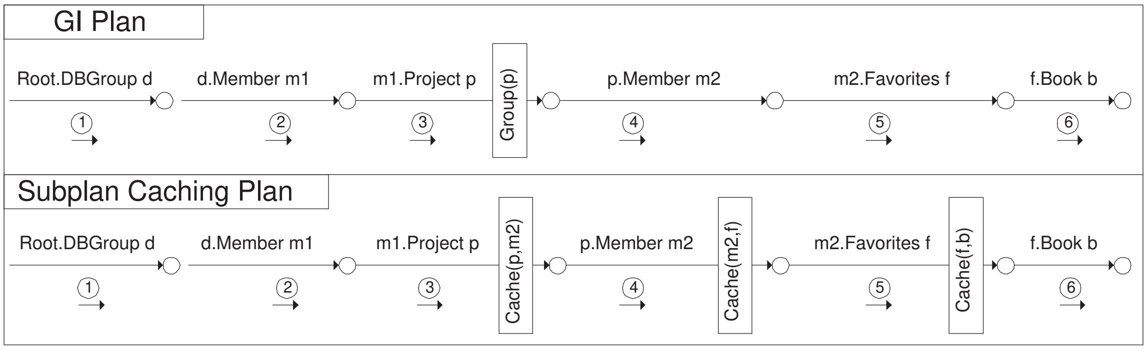

Figure 6.15: Query plan produced for Experiment 6.3.3

experiment were consistent with our expectations. When both optimizations are possible, the GI optimization typically results in a faster query plan than subplan caching. This effect is because GI will never redo work due to duplicate object bindings to a variable, where subplan caching may need to recompute a result due to removal of a cache element.

□

6.4 Related Work

Path expression optimization clearly resembles the access method and join optimization problem in relational databases [SAC+79]. If we view each path expression component (`Book`, `Author`, etc.) as a table, and variable sharing as a join condition, then the vast body of research in the relational model can be applied. There are several reasons why we chose not to simply adapt previous work in the relational model to the problem of optimizing path expressions in Lore:

- Some of the relational work has focused entirely on optimizing join order, without regard to access and join methods, e.g., [GLPK94, IK90, PGLK97, Swa89]. In our setting there is a tight coupling between evaluation order and access methods: some orders preclude certain access methods, and some access methods preclude certain orderings.

- Since we are considering a graph-based data model, pointer-chasing as an access method is typically cheap and supported by low-level storage. Lore also supports inverse pointers via the Lindex. These access methods typically are not supported by relational systems (although they are are similar to join indexes), and have not been considered in relational join order optimization algorithms.

- Path expression optimization benefits from path statistics that are not normally supported by relational systems.

- Some of the relational work has focused on specific "path expression shapes", e.g., *linear queries*, *star queries*, and *branching queries* [OL90]. By contrast, path expressions in Lorel have an arbitrary tree shape.

Heuristic optimization of branching path expressions We explored in Section 6.2 several different heuristics for path expression optimization. In contrast to the algorithms we presented here, relational optimization has considered three major styles of plan space search: exhaustive bottom-up (System-R style), e.g., [OL90, PGLK97, SAC+79]; transformation-based search using iterative improvement or simulated annealing, e.g., [IK90, Swa89]; and random search, e.g., [GLPK94]. We proposed in Section 6.2 a suite of algorithms, each of which reduces the plan space in a different manner and finds the optimal plan within that space. (If we were forced to categorize our algorithms, most of them would be top-down approaches with very aggressive pruning heuristics.) The general ideas underlying most of our algorithms are transferable to the relational setting. Thus, it would be interesting to see the quality of plans generated by our algorithms (appropriately modified) in contrast to those generated by, e.g., [GLPK94, IK90, OL90, PGLK97, Swa89].

The closest work to the algorithms presented in Section 6.2 for object-oriented query optimization is [ODE95], which considers optimizing a restricted form of branching path expressions. Their approach handles a set of linear path expressions where each linear path starts with the same variable, equivalent to relational *branching queries* described in [OL90]. [ODE95] compares exhaustive search with a proposed heuristic search in the context of an object-oriented database system. In both search strategies, cross-products are not considered, and branches are treated as indivisible units in the plans. Our work extends the work of [ODE95] by considering a wider range of path expressions, query plans, and optimization strategies.

Other work on cost-based optimization in object-oriented databases has considered path expressions. [GGT96] optimizes linear path expressions in a two-step process, first by heuristically choosing components of the path expression to be bound using a proposed new

n-ary operator, then using any classical cost-based search strategy to assign the remaining access and join methods. In [SMY90], a dynamic programming algorithm is used to optimize a linear path expression in time $O(n^3)$, where n is the number of classes that appear in the query. Cross-products between classes are not considered and no performance results are reported. The heuristics suggested in both of these papers are not always effective for branching path expressions, so new heuristics for limiting the search space need to be considered.

Grouping introduction [YM98] considers a technique similar to GI in the context of object-oriented databases. They propose pushing a *flatten* operation, which must already exist as part of the query execution plan, down so that the flatten is done as early as possible. In doing so they hope to combine some of the duplicates in a set of sets. For example, if a variable in the query plan is bound to {{o1,o2}, {o2,o3}}, then a flatten operator reduces the number of objects that need to be processed to {o1, o2, o3}. The GI optimization technique is more general since we don't require a flatten operation to be present.

Chapter 7

Views for Semistructured Data

In this chapter we introduce a view management facility for semistructured data. We begin by presenting a view specification language that extends the Lorel query language introduced in Chapter 2. The specification of a view in Lore consists of a sequence of queries and update statements. We then focus on incremental maintenance of materialized views specified in our language. Materialized views replicate objects from the base data and require the view to be made consistent with the base data when it is updated. We present an algorithm to incrementally maintain materialized views and explore when this algorithm is preferred over completely recomputing a view.

The view specification language presented in this chapter appeared originally in [AGM+97]. The work on materialization and maintenance of views appeared originally in [AMR+98].

7.1 Introduction and Motivation

A database *view* is an abstraction of portions of data in a database, suited to a specific user or application. A view is declared by a *view specification language*. The specification is applied over a *source database* (or equivalently *base data*). Database views can be either *virtual* or *materialized*. A virtual view is stored internally as the view specification itself, not the view data which must be computed at query time. A materialized view creates the view data by applying the view specification to the base data and storing the view contents. When the base data changes, materialized views must also be updated, and this process is known as *view maintenance*. View mechanisms have been studied extensively in the context of the relational model [Ull89, KS91] and, more relevant to us, in the context of the object-oriented database model [Ber91, AB91, SLT91, Run92]. This chapter describes a view management facility for semistructured data. We introduce a view specification language that is an extension of Lorel. We also introduce and empirically analyze an algorithm for incremental view maintenance of materialized views. The view specification language and view materialization have been implemented in Lore, however the incremental maintenance algorithm presented here has not been integrated into the Lore views facility.

A unique motivation for views when dealing with semistructured data is that views can be used to introduce some structure into a semistructured database, since a view can group together arbitrary portions of a database into a logical unit. Then, writing and processing a query over a view can potentially be both simpler and more efficient than applying the query to the entire database. For example, consider a large data warehouse stored in Lore that integrates information about millions of people from many heterogeneous sources. In the warehouse a person could be represented by objects with many different structures, but a view would help to present objects in a more structured and regular manner.

Another important motivation for views over semistructured data is that a view mechanism provides a way of creating "stand-alone" databases from the original database. A view can be a subgraph of the original database (perhaps with new objects and new edges added) whose objects can be either replicated (partially or fully) or pointers to objects in the base data. A client/server architecture, where a portion of the database stored at the server is replicated at the client, could utilize this notion of views by treating the replicated information as a view defined over the source.

There are two major difficulties in introducing a view mechanism to a semistructured DBMS. The first difficulty, also found in object-oriented database views, comes from the intermixing of queries and objects. A query in the relational model returns a relation that is by itself a "small" database that makes sense as an independent entity. In contrast, the result of a Lorel query (recall Chapter 2) contains objects that do not have semantics independent of the original database. The second difficulty in introducing a view mechanism to a semistructured database is the absence of a schema. For instance, if we observe the view specification for the ODMG data model [Cat94] used in O2Views [SAD94], the schema, and more precisely the class structure, plays a central role. Since there is no such precise notion of schema for semistructured data, our task is made more difficult.

The remainder of this chapter proceeds as follows. Section 7.2 introduces the view specification language. Details on materialized views, including a complete algorithm to incrementally maintain a materialized view, appear in Section 7.3. Related work is presented in Section 7.9.

7.2 View Specification Language

We introduce a view specification language that is an extension to Lorel. The main goal of the view specification language is to provide a mechanism for *importing* into a view arbitrary objects, and edges between these objects, from a source database. In addition, new objects and edges can be included in the view. A view definition must be able to select entire subgraphs of the source database to be included in the view. To do so, we extend Lore's *select-from-where* queries as defined in Chapter 2 (Section 2.4) to include a *with* clause. The *with* clause allows the user to specify portions of the database (starting from selected objects) that are to be included in the view. More precisely, the *with* clause is made up of path expressions beginning from selected objects (specified by the *select* clause), where each object in the path, along with the edge, is also included in the view. Note that without the *with* clause it would not be possible to directly include subobjects of selected objects along with their edges.

A view specification is composed of an arbitrary sequence of *select-from-where-with* statements as described in the previous paragraph, possibly interleaved with Lorel update statements. Each *select-from-where-with* statement specifies a subgraph of data in the source database that should appear in the view. Update statements allow additional objects and edges to be added to or removed from the view. Queries can be issued over data in a view, over data in the source database, or over both. Views are accessed using path expressions that begin with a unique *name* for that view.

For the examples in this chapter we use data from the Guide database introduced originally in Chapter 2 (Section 2.2). We augment the database with information about entrees that a restaurant serves. The portion of the augmented Guide database used in the examples in this chapter appears in Figure 7.1. Note that we also show a different subset of the restaurants in the Guide database in Figure 7.1 from the restaurants shown in Figure 2.1, although the structure of the data remains the same.

Example 7.2.1 Consider the following example view specification:

```
Define_View MyView as Restaurants =
```

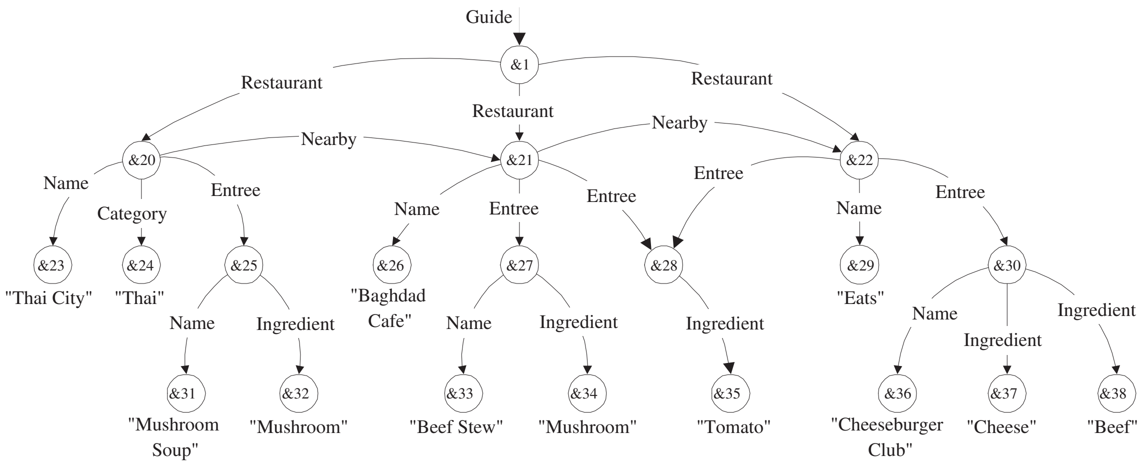

Figure 7.1: Some data for the Guide database

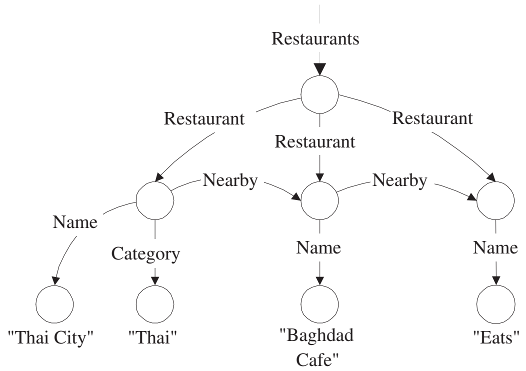

Figure 7.2: View resulting from Example 7.2.1

```
select r
from Guide.Restaurant r
with r.Name, r.Category, r.Nearby
```

This view specification creates a view with the name (entry point) `Restaurants`, which has as subobjects all the restaurant objects along with name, category, and nearby restaurant subobjects if they exist. (`MyView` is used to create a unique "workspace" for the view. Details are not relevant to this chapter.) When the view specification statement is applied to the database in Figure 7.1 it results (logically or physically) in the view shown in Figure 7.2. □

To illustrate in more detail the above view specification, we describe one possible way

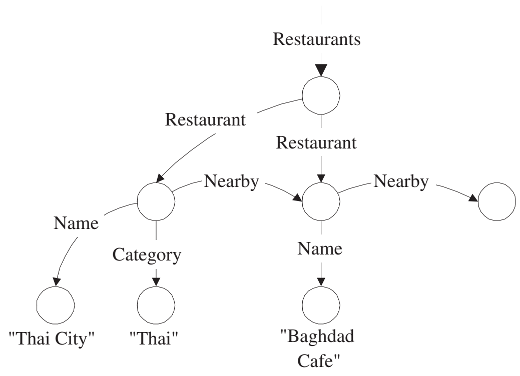

Figure 7.3: View containing unwanted object

that a view could be materialized. Assume a top-down query execution strategy is being used, as described in Chapter 3 (Section 3.3). Evaluation of the *from* and *where* clauses proceeds as in any top-down evaluation of a *select-from-where* query. Then the *select* clause is evaluated, resulting in a set of objects, s, in the source database. Since a materialized view is an independent database graph, each source object in s is replicated to create a corresponding *delegate* object in the view. The rest of the view is constructed as specified by the *with* clause. That is, all possible paths (objects and edges) in the source database that match path expression components in the *with* clause also are copied into the materialized view. In Example 7.2.1, source database objects bound to r will result in delegate objects in the view. Because of the *with* clause, all `Name`, `Category` and `Nearby` edges and objects will also be replicated in the view. Note that each edge and object will only be included once in the view, even if it is reachable by multiple paths in the *with* clause.

The *with* clause does not do object filtering and is applied to all objects that satisfy the *from* and *where* clauses. For instance, suppose we modify the view in Example 7.2.1 to limit the restaurants in the view to those that serve an entree containing mushrooms. We do so by adding the condition `r.Entree.Ingredient = "Mushroom"` to the *where* clause. It is possible that a restaurant R appears in the view because it is "nearby" a restaurant that serves an entree with mushrooms, however R does not satisfy the *where* clause. In such a case, the *with* clause introduces the (possibly unwanted) restaurant R to the view, but does not include any subobjects since R is not bound to r in the query. This is shown by the inclusion of the restaurant object on the far right in Figure 7.3. To completely filter out such restaurants, we can create the view in two steps by first including too much information (as in Figure 7.3), and then removing that which is not needed. This process is illustrated in the following example.

Example 7.2.2 Consider the following view specification consisting of two statements:

```
Define_View MyFavoriteView as Restaurants =
   select r
   from Guide.Restaurant r,
   where r.Entree.Ingredient = "Mushroom"
   with r.Name, r.Category, r.Nearby

   update r.Nearby -= n
   from Restaurants.Restaurant r, r.Nearby n
   where not exists (n.%)
```

This view contains all restaurants that have an entree whose ingredient is mushrooms, as well as edges to nearby restaurants that also serve mushrooms. The result is the same as Figure 7.3 except the unwanted restaurant object on the far right is gone. The above view illustrates the usefulness of multi-statement view specifications. Lorel's update language, originally described in Chapter 2 (Section 2.4.4), is used to specify how edges are (logically or physically) added to or removed from the view. The operator "$-=$" in the update statement indicates that *Nearby* edges between bindings for r and n should not be in the view. □

Note that there are other ways to support creating a view such as the one in Example 7.2.2. For example, we could allow selections in the *with* clause similar to the `having` clause applied after `group by` clauses in relational database management systems. We do not explore this approach here.

7.3 Materialized Views and Maintenance

We now consider the view specification language introduced in Section 7.2 and focus on materialized views and their maintenance. Specifically, we propose and evaluate an algorithm to synchronize a materialized view with base data in the face of base data updates. Our view maintenance algorithm is *incremental*, meaning that we use the base data updates to update as small a portion of the view as possible, instead of recomputing or updating large portions of the view. Limitations to our approach are discussed in Section 7.4. Motivation and preliminary information appears in Section 7.5. Our incremental view maintenance algorithm is presented in Section 7.6. A simple measure to evaluate the effectiveness of our algorithm is presented in Section 7.7. Simulations of the incremental algorithm versus full view recomputation have been performed and are reported in Section 7.8. Our results show that our incremental maintenance algorithm is several orders of magnitude faster than recomputing the view when the base data updates are insertions and deletions of edges between objects. In addition, incremental maintenance is cheaper for small numbers of atomic value changes. However, in some cases, such as when a substantial portion of the database is updated, it may be cost effective to recompute the view.

7.4 Limitations and Notation

The incremental maintenance algorithm presented in this chapter imposes four restrictions on our general view framework:

- A view specification is composed of only a single *select-from-where-with* statement. This statement cannot contain any subqueries in the *where* clauses. Further, the query must be specified by the user in disjunctive normal form (DNF) as described in Chapter 2 (Section 2.4.5). The *select* clause of the *select-from-where-with* statement may contain only a single variable.
- Path expressions in the view specification query may not use regular expression operators or alternation (recall Chapter 2, Section 2.4.3).
- A top-down query execution strategy (recall Chapter 3, Section 3.3) is required for the materialization of a view as well as all queries executed over the view.
- Our incremental maintenance algorithm processes a single update operation at a time.

In this chapter we use a, b, c, x, y, z as variables and L and l to denote labels.

7.5 Motivation and Preliminaries

When materializing a view, our view specification language introduces two types of delegate objects in the view: (1) the *select-from-where* part specifies the *primary* delegate objects of the view, and (2) the *with* part specifies paths from primary objects to *adjunct* delegate objects of the view. During view materialization the *select-from-where* part of the view specification is executed resulting in a set of objects, s, appearing in the source database. One delegate object is created in the view for each object in s; these are the primary objects. From each of the objects in s the path expressions in the *with* clause are evaluated. All objects and edges discovered during this evaluation are also replicated into the view; these objects are the adjunct objects. The distinction between the two kinds of view objects is invisible to the user – it is only used to simplify the discussion of incremental maintenance. Given a view specification and a base data update, our algorithm produces a set of maintenance statements, evaluates them on the database to yield a set of view updates, and installs the updates in the view.

The view specification in Example 7.5.1 below defines a view containing all `Entree` subobjects of a `Restaurant` where the restaurant's name is "Baghdad Cafe" and one of the ingredients of the entree has the value "Mushroom". The view contains the satisfying entrees along with all of their `Name` and `Ingredient` subobjects. We will use this view specification for many of the examples in the remainder of this chapter. Although simple, it serves to illustrate many of the important points.

Example 7.5.1

```
Define_view FavoriteEntrees as Entrees =
  select e
  from Guide.Restaurant r, r.Entree e
  where exists x in r.Name: x = "Baghdad Cafe"
  and exists y in e.Ingredient: y = "Mushroom"
  with e.Name n, e.Ingredient i;
```

Variables are shown explicitly for the path expressions in the *with* clause for ease of presentation. In general, these variables are generated by the system. Figure 7.4 shows the materialized view applied to the database in Figure 7.1. The objects &27, &33, and &34 in Figure 7.1 provide bindings for e, n and i, respectively. The sole primary object &27' and the adjunct objects &33' and &34' are the corresponding delegate objects in the view. Object &99 is the newly created named object for the view. □

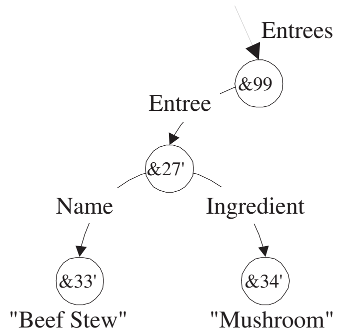

Figure 7.4: The materialized view for Example 7.5.1

7.5.1 Update Operations

For our incremental view maintenance algorithm, all updates to Lore databases are considered at the level of the following three elementary operations:

- Insertion of an edge with label L from the object with oid o_1 to the object with oid o_2, denoted $\langle Ins, o_1, L, o_2\rangle$.
- Deletion of the edge with label L from the object with oid o_1 to the object with oid o_2, denoted $\langle Del, o_1, L, o_2\rangle$.
- Change of the value of the atomic object with oid o_1 from *OldVal* to *NewVal*, denoted $\langle Chg, o_1, \mathit{OldVal}, \mathit{NewVal}\rangle$.

These update primitives capture all updates to the base data that are relevant to view maintenance. If a new object is created, it only becomes relevant when an edges is created connecting it to the database. There is no object deletion operation since (recall from Chapter 2, Section 2.2) objects are never explicitly deleted. Finally, updating the label on an edge is modeled as removing the edge with the old label and then adding a new edge with the new label.

7.6 View Maintenance Algorithm

When an update operation to base data potentially affects a materialized view, the view may need to be modified to keep it consistent with the database. A view V is considered *consistent* with the database DB if the evaluation of the view specification S over the database yields the view instance $V = S(DB)$. Therefore, when the database DB is updated to DB', we need to update the view V to $V' = S(DB')$ in order to preserve its

1. View specification statement S:
   select v_i
   from $v_0.L_1\ v_1,\ \ldots,\ v_j.L_k\ v_k,\ \ldots,\ v_{n-1}.L_n\ v_n$
   // v_x can be any variable that already appeared in the sequence or a name
   where conditions$(v_1, \ldots, v_n)$
   with $v_i.L_{11}\ w_{11},\ w_{11}.L_{12}\ w_{12},\ \ldots,\ w_{1(p-1)}.L_{1p}\ w_{1p},$
   $u_j.L_{j1}\ w_{j1},\ \ldots,\ w_{j(k-1)}.L_{jk}\ w_{jk},\ \ldots,\ w_{j(q-1)}.L_{jq}\ w_{jq}$
   // where u_j is v_i or w_{kl} $(2 \leq j,\ 1 \leq k \leq (j-1),\ 1 \leq l)$
2. Update U: $\langle Ins, o_1, L, o_2\rangle$, $\langle Del, o_1, L, o_2\rangle$, or $\langle Chg, o_1, OldVal, NewVal\rangle$
3. New database state DB'
4. View instance V

Figure 7.5: Incremental maintenance algorithm input

consistency.

Our incremental maintenance algorithm computes the new state of a materialized view from the current (post-update) state of the database, the view, and the database updates. Similar to relational view maintenance algorithms, the incremental maintenance algorithm uses the database updates to minimize the portion of the database examined when computing the view updates [GM95].

7.6.1 Overview of the Maintenance Algorithm

Our overall maintenance algorithm is divided into a number of procedures shown in Figures 7.7, 7.8, 7.9, 7.11, and 7.12. The input to the algorithm and the basic steps are shown in Figure 7.5 and Figure 7.6. Note that in Figure 7.5 we abbreviate the *where* clause as "conditions$(v_1, \ldots, v_n)$." As discussed in Section 7.4 the *where* clause must be in disjunctive normal form. We treat the primary and adjunct view objects (V_{prim} and V_{adj}) separately during maintenance.

The view specification S, the database update U, and the database state DB' after the update are used to compute a sequence of *view maintenance statements* in Lorel. We needed to extend Lorel to allow the use of explicit object identifiers wherever names or variables are allowed within a statement. The maintenance statements generate sets ADD_{prim}, DEL_{prim}, ADD_{adj}, and DEL_{adj} of objects and edges to add to and remove from the view. After the sets ADD_{prim}, DEL_{prim}, ADD_{adj}, and DEL_{adj} are generated we must *install* the changes by adding and removing objects and edges in the view.

Figure 7.6 outlines the steps of the view maintenance algorithm. Details will be provided

1. Check for relevance of update U to the view instance V defined by the view specification S. Generate a set of *relevant variables* R. If R is empty, stop.
2. Generate maintenance statements and create ADD_{prim} and DEL_{prim} using U, S, and R.
3. Generate maintenance statements and create ADD_{adj} and DEL_{adj} using U, S, R, and ADD_{prim} or DEL_{prim}.
4. Install ADD_{prim}, DEL_{prim}, ADD_{adj}, and DEL_{adj} in V.

Figure 7.6: Basic steps of the incremental maintenance algorithm

below. We describe the algorithm as it operates on a single update. First, it checks whether the update is *relevant* to the view, that is, if update U could potentially cause a change to the view instance V. It does this by generating a set of variables, the *relevant variable set*, where each variable in the set could be bound to an object involved in the update operation. If this set is non-empty then the algorithm creates the Lorel maintenance statements that generate ADD_{prim} and DEL_{prim}. These statements identify primary objects to add to and remove from the view by explicitly binding the objects in the update to the view specification. The algorithm then creates the Lorel maintenance statements that generate ADD^x_{adj} and DEL^x_{adj}. ADD^x_{adj} and DEL^x_{adj} contain the adjunct objects and edges to add to and remove from the view for a variable x appearing in a path expression component for a path expression in the *with* clause. Adjunct objects may be affected in three ways: (1) by newly inserted or deleted primary objects; (2) by current adjunct objects that are affected by an inserted or deleted edge in the base data; and (3) by atomic value changes.

7.6.2 Relevance of an Update

To avoid generating (and evaluating) unnecessary maintenance statements, we first perform some simple relevance checks. For each view, we maintain an auxiliary data structure, *RelevantOids*, to keep information that would be available from the schema in a structured database. *RelevantOids* contains the object identifier of every object touched during the evaluation of the view specification, paired with the variable to which it was bound, whether or not the object eventually appears in the view. This information is used to check quickly whether a database update could possibly affect the view. For example, if object o_1 in a *Chg* update does not appear in *RelevantOids*, then o_1 was not examined during view evaluation and the update can safely be ignored.

function *RelevantVars*(Update U, View specification S)$\Rightarrow$ SetOfVariables
// *RelevantOids* is $\{\langle oid, queryvariable \rangle\}$. $o_1(U)$ returns the first oid in the update structure.
// The update structure is defined in Figure 7.5.
if $\langle o_1(U), \cdot \rangle \notin$ *RelevantOids* **then return** ϕ;
// Find out which variables are relevant to the update
$vars \leftarrow \phi$; $relvars \leftarrow \phi$;
foreach $v \in$ variables(S) **do**
    // If updated object is not in *RelevantOids*, then it's not relevant
    if $\langle o_1(U), v \rangle \in$ *RelevantOids* **then** $vars \leftarrow vars \cup \{v\}$;
// If update is atomic change, do simple syntactic check
if type(U) $= Chg$ **then**
    foreach $v \in vars$ **do**
        // Let constants(S, op, v) be the constants appearing in S compared to v
        // with comparison operator op
        foreach $c \in$ constants(S, op, v) **do**
            // See if there's a predicate in the view spec whose value may have changed
            if (!*OldVal*(U) op c **and** *NewVal*(U) op c) **or**
                (*OldVal*(U) op c **and** !*NewVal*(U) op c) **then**
                $relvars \leftarrow relvars \cup \{v\}$;
else $relvars \leftarrow vars$;
return $relvars$;

Figure 7.7: Pseudocode for the *RelevantVars* algorithm.

RelevantOids must be updated when the view is updated, which may involve adding new objects to *RelevantOids*. New relevant objects include all objects touched during evaluation of the maintenance statements. Maintaining *RelevantOids* could also involve the removal of objects. However, it is not easy to decide when to remove an object, since an object may be relevant because of multiple paths through the source database. Instead, we ignore potential removals and let *RelevantOids* contain a superset of relevant objects. This approach may lead to unnecessary maintenance statements but always results in a consistent view. If many deletions from the view cause *RelevantOids* to remain artificially large, *RelevantOids* may be recomputed during a time of low system load. We also use syntactic checks that indicate whether specific atomic value changes could affect the view. For each comparison in the view specification *where* clause that involves a constant value, we compare the constant to the update's *OldVal* and *NewVal*. If both or neither of *OldVal* and *NewVal* satisfy the comparison for all disjuncts in the *where* clause, then the change cannot affect the view.

Figure 7.7 presents the function *RelevantVars*, which returns the set of variables appearing in a view specification query that can be affected by a given base data update. As an example, suppose that the value of object &23 in Figure 7.1 is changed from "Thai City" to "Hunan Wok". We can infer that this update cannot affect the view in Example 7.5.1, because the view specification mentions neither "Thai City" nor "Hunan Wok". On the other hand, if the value of &23 is changed to "Baghdad Cafe", which is the constant used in the comparison *x.Name = "Baghdad Cafe"*, then the update may be relevant. As another example, consider the materialization of the view given in Example 7.5.1 over the Guide database shown in Figure 7.1. The *RelevantOids* structure contains $\langle$&30, $e\rangle$ since even though &30 is not part of the view, it was visited during the materialization of the view. If the update operation $\langle Ins,$ &30, `Ingredient`, &34$\rangle$ occurs then the first `foreach` loop of *RelevantVars* results in e being added to the result set. Since the update operation is not an atomic value change, the second `if` statement in Figure 7.7 fails and *RelevantVars* returns $\{e\}$.

We do not attempt to quantify the savings achieved by using *RelevantOids* and *RelevantVars* in this work. However, we note that for views defined over a small portion of the database, most updates are irrelevant.

7.6.3 Generating Maintenance Statements

We now describe how to generate the maintenance statements for each type of update: edge insertion, edge deletion, and atomic value change. Consider first the edge insertion and edge deletion cases. For each path expression component in the view specification, we generate a maintenance statement that checks whether the updated edge can bind to it. If so, the statement produces updates to the view. We use auxiliary data structures to represent the components appearing in the view specification. $Component_{from}$, $Component_{prim}$, and $Component_{adj}$ contain all the path expression components that appear in the *from* clause, *from* and *where* clauses, and *with* clause, respectively. For example, $Component_{prim}$ for the view specification in Example 7.5.1 is $\{$`Guide.Restaurant` r, r.`Entree` e, r.`Name` x, e.`Ingredient` $y\}$. Note that each *Component* set is small since it depends on the query and not on the database.

Edge Insertion. For edge insertion, let the update be $\langle Ins, o_1, L, o_2\rangle$. We generate a primary object maintenance statement for every possible pair of bindings of o_1 and o_2 to variables identified by *RelevantVars* using the procedure *GenAddPrim* in Figure 7.8.

procedure *GenAddPrim*(Update $U = \langle Ins, o_1, L, o_2\rangle$, View specification S, *RelevantVars* R)
// For each relevant variable
foreach $a \in R$ **do**
  // For each place where the update can be substituted in the view specification
  foreach $\langle a,L,b\rangle \in Component_{prim}$ **do**
    // Write a maintenance statement based on the view specification
    ADD_{prim}+= copy S except for the *with* clause and
      $\forall L_i$ in *from* clause replace "$a.L_i$" with "$o_1.L_i$"
      $\forall L_j$ in *from* clause replace "$b.L_j$" with "$o_2.L_j$"
      replace "a" with "o_1" and "b" with "o_2" in *where* clause
      add "**and** $a = o_1$" to each disjunct in *where* clause
      if $\langle a,L,b\rangle \in Component_{from}$
        add "**and** $b = o_2$" to each disjunct in *where* clause

Figure 7.8: Pseudocode for the *GenAddPrim* algorithm

Example 7.6.1 (Generating ADD_{prim}) Suppose that update $\langle Ins, \&28, Ingredient, \&34\rangle$ is performed on the database in Figure 7.1. The Baghdad Cafe restaurant now has two entrees with the ingredient "Mushroom". Given the view specification in Example 7.5.1, *RelevantVars* returns the set {e}. *GenAddPrim* then generates one statement:

ADD_{prim} += **select** e
  from Guide.Restaurant r, r.Entree e
  where exists x **in** r.Name: x = "Baghdad Cafe"
  and **exists** &34 **in** &28.Ingredient: &34 = "Mushroom"
  and e = &28;

Note that in this example, we did not need to say "**and** y = &34," since the variable y is not part of a path component in the *from* clause. The result of this query, {&28}, is added to the set ADD_{prim}. This maintenance query can be evaluated more efficiently than the original view specification, as we show in Section 7.7. □

We then generate the maintenance statements for the adjunct objects. There are two cases of how adjunct objects can be added to the view: (1) adjunct objects attached to the new primary objects in ADD_{prim} and (2) adjunct objects that are newly connected to the view because the delegate object for the parent object for the inserted edge appears in the view.

For the first case, we generate maintenance statements starting from the set ADD_{prim}. For the second case, we first test whether the inserted edge matches a relevant variable and has a matching label. If so, then we generate a set of maintenance statements that add

```
procedure GenAddAdj(Update U = ⟨Ins, o1, L, o2⟩, View specification S, RelevantVars R)
// (1) If primary objects were added, need to add adjunct objects from them
if ADD_prim ≠ φ then
   // For each path component in the with clause
   foreach ⟨w_j(k-1),L_jk,w_jk⟩ ∈ Component_adj do
      // Write a maintenance statement based on view specification (no where or with clause)
      ADD_adj^w_jk += select ⟨w_j(k-1),L_jk,w_jk⟩
                      from  ADD_prim v_i, v_i.L_j1 w_j1, ..., w_j(k-1).L_jk w_jk;
// (2) For each place that edge could be adjunct edge
foreach v ∈ R do
   foreach ⟨v,L,w_jk⟩ ∈ Component_adj do
      // Write a set of maintenance statements starting from added edge:
      // Add inserted edge to the view
      ADD_adj^w_jk += select ⟨o1,L,o2⟩;
      // From o2, add any necessary edges
      ADD_adj^w_j(k+1) += select ⟨o2,L_j(k+1),w_j(k+1)⟩ from o2.L_j(k+1) w_j(k+1);
      // In a similar fashion, include all paths
      foreach ⟨w_j(k+n-1),L_j(k+n),w_j(k+n)⟩ ∈ O do
         ADD_adj^w_j(k+n) += select ⟨w_j(k+n-1),L_j(k+n),w_j(k+n)⟩
                             from  o2.L_j(k+1) w_j(k+1), ..., w_j(k+n-1).L_j(k+n) w_j(k+n);
```

Figure 7.9: Pseudocode for the *GenAddAdj* algorithm

the inserted edge and all subsequent paths in $Component_{adj}$. Both cases are handled by procedure *GenAddAdj* in Figure 7.9.

Example 7.6.2 (Generating ADD_{adj}) *GenAddAdj* generates the following maintenance statements for the update ⟨Ins, &28, Ingredient, &34⟩.

ADD^n_{adj} += **select** ⟨*e*,*Name*,*n*⟩
from ADD_{prim} e, e.Name n;

ADD^i_{adj} += **select** ⟨*e*,*Ingredient*,*i*⟩
from ADD_{prim} e, e.Ingredient i;

Since the inserted edge for this example is not connected to an object with an existing delegate adjunct object (&28 has no delegate in the view), we consider only case (1) in *GenAddAdj*. Notice that the statements above do not directly operate over the view (like a Lorel update statement operates over data), rather they identify objects and edges that will be added to the view. We discuss the installation of these changes, which will result in added objects and edges to the view, in Section 7.6.4. □

Because the addition of an edge in the absence of negation cannot cause a deletion, we do not have to generate DEL_{prim} or DEL_{adj}. After installing both ADD_{prim} and ADD_{adj},

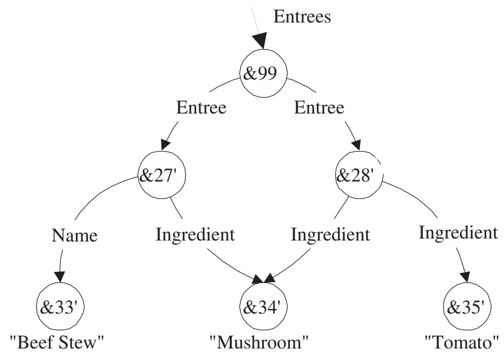

Figure 7.10: New view instance after update ⟨Ins, &28, Ingredient, &34⟩

Figure 7.10 shows the new instance of the view for the given example update.

Edge Deletion. Let the update be $\langle Del, o_1, L, o_2\rangle$. A deleted edge may: (1) be irrelevant and not affect the view; (2) cause a primary view object (or objects) to be deleted; or (3) may have a corresponding edge in the view that needs to be removed. Either (2) or (3) could cause additional edges to adjunct objects to be removed from the view. In principle, a delete edge update generates maintenance statements similar to an insert edge update. However, the delete edge statements must simulate the existence of the now deleted edge in the base data since maintenance of a view is performed after the database update. We must simulate the existence of the removed edge to determine whether it originally contributed to the appearance of objects or edges in the view. Also, the delete edge statements must check (using a subquery) whether a potentially deleted object or edge should remain in the view due to paths not involving the deleted edge. For example, if the *Entree* object &27 in Figure 7.1 had *two* "Mushroom" ingredients, then applying the update $\langle Del, \&27, Ingredient, \&34\rangle$ should not remove the *Entree* object &27 from the view.

One solution to the problem of maintaining a consistent view in the case of edge deletion is to maintain *counts* of the numbers of derivations for each view object, following the spirit of [GMS93]. However, the dynamic maintenance of these counts can be prohibitively expensive, because the derivations of many view objects may depend on one edge. We use a nested subquery that will not remove primary objects when the object remains in the view after the update operation has been performed. Figure 7.11 shows the procedure *GenDelPrim*, used to generate the maintenance statements for the primary objects.

Example 7.6.3 (Generating DEL_{prim}) Suppose the update $U = \langle Del, \&21, Entree, \&27\rangle$

```
procedure GenDelPrim(Update U = ⟨Del, o1, L, o2⟩, View specification S, RelevantVars R)
// For each relevant variable
foreach a ∈ R do
   // For each place where the update can be substituted in the view specification
   foreach ⟨a,L,b⟩ ∈ Component_prim do
      // Write a maintenance statement based on the view specification:
      DEL_prim += copy S except for the with clause and
            // The first two replacements reconstruct the before state
            replace "a.L b" with "(o1.L ∪ {o2}) b" in from clause.
            replace "exists b in a.L" with "exists b in (o1.L ∪ {o2})" in where clause.
            // The remaining handle normal appearance of bound variables
            ∀L_i in from clause replace "a.L_i" with "o1.L_i"
            ∀L_j in from clause replace "b.L_j" with "o2.L_j"
            replace "a" with "o1" in where clause
            replace "b" with "o2" in where clause
            add "and a = o1" to each disjunct in where clause
            if ⟨a,L,b⟩ ∈ Component_from
               add "and b = o2" to each disjunct in where clause
            // The duplicate test is a subquery that ensures that the object
            // bound to v_i is not in the view for another reason
            add to where clause "and not exists (S')" where S' is
              S without the with clause and with new variables v'_j for each v_j
              and an additional condition: "v'_i = v_i" (v_i is the selected variable in S)
```

Figure 7.11: Generating maintenance statements for DEL_{prim}

Clause	Original	Incremental Statement	General Rule
From	Guide.Restaurant r	Guide.Restaurant r	$v_j.L_k\ v_k$ such that ($v_j \neq a$ and $v_j \neq b$ and $v_k \neq a$ and $v_k \neq b$) $\rightarrow$ No Change
From	r.Entree e	(&21.Entree $\cup$ {&27}) e	$a.L\ b \rightarrow (o_1.L \cup \{o_2\})\ b$
Where	$\exists x$ **in** r.Name: $x =$ "Baghdad Cafe"	$\exists x$ **in** &21.Name: $x =$ "Baghdad Cafe"	$a.L_j\ v_j$ such that $v_j \neq b \rightarrow o_1.L_j\ v_j$
Where	$\exists y$ **in** e.Ingredient: $y =$ "Mushroom"	$\exists y$ **in** &27.Ingredient: $y =$ "Mushroom"	$b.L_j\ v_j \rightarrow o_2.L_j\ v_j$

Table 7.1: Transformations for maintenance statements for Example 7.6.3

Clause	Original	Incremental Statement	General Rule
From	$e.Price\ p$	$\&27.Price\ p$	$b.L_k\ v_k \rightarrow o_2.L_k\ v_k$
Where	$\exists e$ **in** $r.Entree$: ...	$\exists e$ **in** $(\&21.Entree \cup \{\&27\})$	$a.L\ b \rightarrow (o_1.L \cup \{o_2\})\ b$

Table 7.2: Additional transformation rules for Example 7.6.3

is applied to the database of Figure 7.1. The object &27 must be removed from the view. *GenDelPrim* generates the following statement:

DEL_{prim} += **select** e
from Guide.Restaurant r, (&21.Entree ∪ {&27}) e
where exists x **in** &21.Name: x = "Baghdad Cafe"
and **exists** y **in** &27.Ingredient: y = "Mushroom"
and r = &21 **and** e = &27
and **not exists** (
select e′
from Guide.Restaurant r', r'.Entree e'
where exists x' **in** r'.Name: x' = "Baghdad Cafe"
and **exists** y' **in** e'.Ingredient: y' = "Mushroom"
and e' = e);

This statement adds bindings for r and $r.Entree$ to the view specification S and reconstructs the deleted edge by binding e to &27. The transformations to the original query are summarized in Table 7.1. Note in Table 7.1 that a and b are variables that are members of the sets of variables returned by *RelevantVars* when o_1 and o_2 (respectively) are used as input. □

Two additional rules, given in Table 7.2, handle situations that are not illustrated by Example 7.6.3. In the first rule in Table 7.2, b feeds a path expression component appearing in the *from* clause. In this situation variable b is replaced with object o_2. In the second rule, variables a and b both appear in the *where* clause and are replaced with objects o_1 and o_2 respectively. This case is almost identical to both o_1 and o_2 in the *from* clause.

After generating DEL_{prim}we generate the maintenance statements for the adjunct objects and the edges leading to them (hereafter called adjunct edges). Since an adjunct object or edge can be included in the view due to multiple paths we cannot delete an adjunct object or edge based on reachability alone. Thus, again a subquery of the *where* clause looks for other variable bindings for the edge to be removed. If another binding is found, then the edge is not deleted. Procedure *GenDelAdj* in Figure 7.12 generates the maintenance statements for the adjunct objects and edges.

Example 7.6.4 (Generating DEL_{adj}) For the update $\langle Del, \&21, Entree, \&27\rangle$, procedure *GenDelAdj* creates one maintenance statement for each path in $Component_{adj}$.

DEL^n_{adj} += **select** $\langle e, Name, n\rangle$
from DEL_{prim} e, e.Name n;

DEL^i_{adj} += **select** $\langle e, Ingredient, i\rangle$
from DEL_{prim} e, e.Ingredient i;

For this example the adjunct objects in the view are affected by the removal of primary objects. Therefore, in Figure 7.12, the first `if` statement evaluates to *true*. The first `foreach` statement executes twice, once for each path expression component appearing in the *with* clause. Each pass through the `foreach` generates a single maintenance statement that identifies, starting from primiary objects that are to be removed from the view, adjunct objects that also must be removed. Note that neither statement in our simple example includes a *where* subclause. Since the label `Entree` does not appear in the path expression components for the *with* clause of the view specification, no work is done by the nested `foreach` statements in Figure 7.12 that appear after the first `if` statement. *GenDelAdj* thus results in the two maintenance statements shown above. □

Atomic Value Change. Let the update U be $\langle Chg, o_1, OldVal, NewVal\rangle$. Of course a value change made to an object in the source database must be propagated if there is a corresponding delegate object in the view. This operation is supported by mapping from base data objects to their delegates in the view. In addition, a value change may cause edge deletions, edge insertions, or both to the view, or there might be no update necessary because the change is irrelevant to the view. Due to object sharing, an object may have many incoming edges with different labels. Therefore, the original edge traversed to find an object is not the only possible relevant edge. Consequently, we bind the changed object to each variable identified by the procedure *RelevantVars*, using a separate maintenance statement for each variable. When an atomic value change could cause the addition of objects to the view, the inserted edge is the edge followed to get to the atomic object

```
procedure GenDelAdj(Update U = ⟨Del, o1, L, o2⟩, View Specification S, RelevantVars R)
if DEL_prim ≠ φ then
   // Deletion of primary objects could affect every path component in the adjunct.
   // Identify edges that need to be deleted because of the removal of primary objects:
   foreach ⟨w_j(k-1),L_jk,w_jk⟩ ∈ Component_adj do
      // Write one maintenance statement for each path component in the with clause.
      // The path in the from clause has to match some path, starting at the selected
      // variable v_i, in the with clause of the view specification (see Figure 7.5).
      DEL_adj^w_jk += select ⟨w_j(k-1),L_jk,w_jk⟩
                      from   DEL_prim v_i, v_i.L_j1 w_j1, ..., w_j(k-1).L_jk w_jk
                      where not exists (
                             select true
                             // the where clause contains one subclause for each path in the
                             // with clause of the view specification that leads to a variable
                             // that uses label L_jk
                             where (V_prim v_i, v_i.L_j1 w'_j1, ..., w'_j(k-1).L_jk w'_jk,
                                   and w'_j(k-1) = w_j(k-1) and w'_jk = w_jk)
                             or    ...);

// An adjunct object could be affected if the label of the deleted edge appears in
// the with clause
foreach u ∈ R do
   foreach ⟨u,L,u_i⟩ ∈ Component_adj do
      // Ensure that u_i is relevant with respect to o2
      if ⟨o2, u_i⟩ ∈ RelevantOids then
         // Must remove the edge from the view since it is deleted from the database
         DEL_adj^u_i += select ⟨o1,L,o2⟩;
         // Write a set of maintenance statements "starting" from the deleted edge to delete
         // all the edges in the view along paths that start from variable u_i for u_i = o2
         foreach ⟨w_j(k-1),L_jk,w_jk⟩ ∈ Component_adj do
            // The path in the from clause has to match some path starting at u_i in
            // the with clause of the view specification statement (see Figure 7.5)
            DEL_adj^w_jk += select ⟨w_j(k-1),L_jk,w_jk⟩
                            from   o2.L_j1 w_j1, ..., w_j(k-1).L_jk w_jk
                             // Same subquery as above
                            where not exists (
                                   select true
                                   where (V_prim v_i, v_i.L_j1 w'_j1, ..., w'_j(k-1).L_jk w'_jk,
                                         and w'_j(k-1) = w_j(k-1) and w'_jk = w_jk)
                                   or    ...);
```

Figure 7.12: Generating maintenance statements for DEL_{adj}

```
procedure GenAtomic(Update U = ⟨Chg, o1, OldVal, NewVal⟩, View S, RelevantVars R)
// For each relevant variable
foreach v ∈ R do
   // Check the condition to see whether atomic value change could cause insertions
   // or deletions in the view for variable v. Constants(S, op, v) is defined in Figure 7.7.
   // The function IncrementalMaint is the overall incremental maintenance algorithm
   // as shown in Figure 7.6.
   // OemNill is a special OEM object that is not equal to itself or any other object
   foreach c ∈ constants(S, op, v) do
      if (!OldVal(U) op c and NewVal(U) op c) then
         IncrementalMaint(⟨Ins, OemNill, l, o1⟩, S)
      if (OldVal(U) op c and !NewVal(U) op c) then
         IncrementalMaint(⟨Del, OemNill, l, o1⟩, S)
```

Figure 7.13: Pseudocode for the *GenAtomic* algorithm

during view evaluation. Conversly, when an atomic value change could cause the deletion of objects in the view, the deleted edge is the edge followed to get to the atomic object during view evaluation. Note that *RelevantVars* can help optimize the execution of the maintenance statements by tracking whether the changed value could potentially cause the addition or removal of objects. In the final `if` statement in *RelevantVars*, shown in Figure 7.7, when the old value matches and the new value does not match then we treat the atomic value change as an edge deletion. If the old value does not match, but the new value matches then we treat the atomic value change as an edge insertion. This logic appears in the procedure *GenAtomic* in Figure 7.13, which is the procedure that generates the maintenance statements for atomic value updates.

Example 7.6.5 (Atomic Value Change) Suppose the update U is ⟨Upd, &26, "Baghdad Cafe", "Wendy's"⟩. The procedure *RelevantVars* identifies x as the only relevant variable for the view specification given in Example 7.5.1: x is bound to object &26 during view materialization and object &26's value before the update is "Baghdad Cafe". This atomic value change cannot result in adding new objects to the view, because the new value "Wendy's" does not satisfy the condition on x. If x is bound to &26 then the condition's value changes from true to false and some objects may no longer be in the view. We therefore generate DEL_{prim} for the deletion of ⟨r,*Name*,&26⟩ since *Name* is the label used in the path expression component that contains x as a destination variable.

DEL_{prim} += **select** e
from Guide.Restaurant r, r.Entree e
where exists &26 **in** r.Name : (*OldVal*(&26) = "Baghdad Cafe")
and **exists** y **in** e.Ingredient: y = "Mushroom"
and **not exists** (
select e'
from Guide.Restaurant r', r'.Entree e'
where exists x' **in** r'.Name: x' = "Baghdad Cafe"
and **exists** y' **in** e'.Ingredient: y' = "Mushroom"
and e' = e);

Based on DEL_{prim}, DEL^n_{adj} and DEL^i_{adj} are calculated as shown by algorithm *GenDelAdj* shown in Figure 7.12. □

7.6.4 Installing the Maintenance Changes

The changes represented by ADD_{prim}, ADD_{adj}, DEL_{prim}, and DEL_{adj} must be installed in the materialized view. Since there is no duplication of objects in the view, deletions need to be installed in the view before insertions. If a view object ceases to be a primary object, it may still remain in the view as an adjunct object and vice versa. Finally, if the update is an atomic value change of an object in the view, the new value is installed in the delegate object.

7.7 Cost Model

We now present an analytic model that estimates the cost of complete view recomputation versus incremental maintenance for a given update. The formulas in the model are based on the database statistics introduced in Chapter 3 (Section 3.4.3). Recall that for views we only consider top-down query execution strategies for maintenance statements and for initial view materialization. As in the cost model for Lore's general query execution engine (Chapter 3), the cost assigned to a view materialization or maintenance statement is the estimated number of object fetches required for execution of the statement.

The formulas use $Fout(x, L)$, which returns the estimated number of L-labeled subobjects for objects bound to x, and $|x|$, which returns the estimated number of objects to be

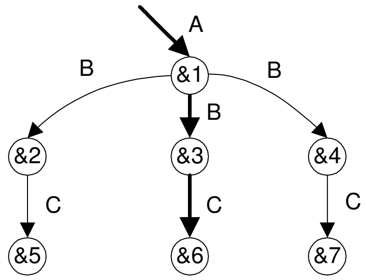

Figure 7.14: Path expression evaluation and statistics

bound to x. Both *Fout* and $|x|$ were defined in Chapter 3 (Section 3.4.3). For our purposes in this chapter we introduce $Cost(x, L, y)$, which returns the estimated cost to get all of the L-labeled subobjects of any object bound to x: $Cost(x, L, y) = |x| * Fout(x, L)$. Note that this function is similar to the cost for a *Scan* operator (Chapter 3, Section 3.4.3) except *Cost* considers together all objects that will be bound to variable x. As an example, given the path expression $b.C\ c$ of Figure 7.14, the evaluation cost for $b.C\ c$ is $Cost(b, C, c) = |b| * Fout(b, C) = |A| * Fout(A, B) * Fout(b, C) = 1 * 3 * 1 = 3$, where A is a named object.

Given the statistics and *Cost* function shown above, we now present the formula used to estimate the cost of evaluating a view specification and the cost of maintenance statements. Our cost formula is much simpler than the generic formulas introduced in Chapter 3 because we consider I/O cost only. The cost for executing any maintenance or view specification statement is simply the cost of evaluating all path expressions in a top-down query strategy in the statement. Without any bindings, the cost for evaluating a path expression P given a top-down query execution strategy is

$$Cost(total) = \sum_{\langle x,L,y\rangle \in P} Cost(x, L, y)$$

The incremental maintenance statements bind variables to the objects contained in the update and use the bindings to prune the search space. The execution proceeds until a variable x bound by the update is encountered. If the object bound to x is not the updated object, then the evaluation short circuits and goes on to the next binding for x. A bound variable lowers the cost of the computation for the rest of the path expression since it limits the remaining portion of a path to objects reachable from the bound variable. In Figure 7.14, consider the path expression "`A.B b, b.C c`" and suppose the binding

$b = \{\&3\}$ is given. The valid set of objects for c, given the binding for b, is $\{\&6\}$. The single valid evaluation of the path expression is shown in bold in Figure 7.14. Without binding b, c would evaluate to $\{\&5, \&6, \&7\}$. Thus the evaluation of path expression "`A.B b, b.C c`" without the binding for b results in 7 object fetches. With the binding for b the cost is reduced to 5 object fetches. Now suppose that c is bound. The cost of executing the first path component is $|A| * Fout(A, B)$. Once we have a B subobject of the named object A, we know all of the subobjects of B since their oid's are stored with the `B` subobject (recall Chapter 2). Thus, if c is bound, the relevant C subobject can be identified from the B object. In this case, the total cost of evaluating the path expression is simply $1 + |A| * Fout(A, B)$ $+ 1$. Insertions and deletions provide two object bindings, while an atomic value change provides only one.

Example 7.7.1 (Cost of Full Recomputation) The cost for full recomputation of the view specification given in Example 7.5.1 is:

$$\begin{aligned}
&\sum_{\langle x,L,y\rangle \in Component_{prim} \cup Component_{adj}} Cost(x, L, y) \\
&= |Guide| * Fout(Guide, Restaurant) * (1 + Fout(r, Entree) + Fout(r, Name) + \\
&\quad Fout(r, Entree) * Fout(e, Name) + 2 * Fout(r, Entree) * Fout(e, Ingredient))
\end{aligned}$$

□

We now show how our cost formula applies to the maintenance statements produced by the update $\langle Ins, \&28, Ingredient, \&34\rangle$ in Example 7.6.1.

Example 7.7.2 (Maintenance Cost of Inserting an Edge) $P = \{\langle Guide, Restaurant, r\rangle$, $\langle r, Entree, e\rangle$, $\langle r, Name, x\rangle$, $\langle e, Ingredient, y\rangle\}$ is the set of path components in the maintenance statement of Example 7.6.1 and $P' = \{\langle ADD_{prim}, Name, n\rangle$, $\langle ADD_{prim}, Ingredient, i\rangle\}$ is the set of path components in the maintenance statement of Example 7.6.2. The bindings $e =$ $\&28$ and $y = \&34$ are provided.

$$\begin{aligned}
&\sum_{(x,L,y)\in P} Cost(x, L, y) + \sum_{(x,L,y)\in P'} Cost(x, L, y) \\
&= |Guide| * Fout(Guide, Restaurant) + 1 + \\
&\quad |Guide| * Fout(Guide, Restaurant) * \\
&\quad Fout(r, Name) + 1 * Fout(e, Ingredient) +
\end{aligned}$$

$$|ADD_{prim}| * (Fout(ADD_{prim}, Name) +$$
$$Fout(ADD_{prim}, Ingredient))$$

$|ADD_{prim}|$ depends upon the number of possible bindings for e and the selectivity of the *where* clause, as follows:

$$|ADD_{prim}| = |e| * Selectivity(where) = 1 * Selectivity(where) \leq 1.$$

□

Finally, we apply the cost formulas to the atomic value change maintenance statements shown in Example 7.6.5.

Example 7.7.3 (Maintenance Cost of an Atomic Value Change) Recall that the incremental maintenance statement for Example 7.6.5 is executed with the binding {&26} for x. $Fout(r, Name)$ is therefore reduced to 1. The cost formula for the incremental maintenance statement for Example 7.6.5 is:

$$\begin{aligned}
&\sum_{(x,L,y)\in P} Cost(x, L, y) + \sum_{(x,L,y)\in P'} Cost(x, L, y)\\
&= Cost(Guide, Restaurant, r) + Cost(r, Entree, e) + Cost(r, Name, x) +\\
&\quad Cost(e, Ingredient, y) + Cost(Guide, Restaurant, r') + Cost(r', Entree, e') +\\
&\quad Cost(r', Name, x') + Cost(e', Ingredient, y') + Cost(DEL_{prim}, Name, n) +\\
&\quad Cost(DEL_{prim}, Ingredient, i)\\
&= |Guide| * Fout(Guide, Restaurant) + |r| * Fout(r, Entree) + |r| *\\
&\quad Fout(r, Name) + |e| * Fout(e, Ingredient) +\\
&\quad |Guide| * Fout(Guide, Restaurant) +\\
&\quad |r'| * Fout(r', Entree) + |r'| * Fout(r', Name) + |e'| * Fout(e', Ingredient) +\\
&\quad |DEL_{prim}| * Fout(DEL_{prim}, Name) + |DEL_{prim}| * Fout(DEL_{prim}, Ingredient)\\
&= |Guide| * Fout(Guide, Restaurant) + |Guide| * Fout(Guide, Restaurant) *\\
&\quad Fout(r, Entree) + |Guide| * Fout(Guide, Restaurant) * Fout(r, Name) +\\
&\quad |Guide| * Fout(Guide, Restaurant) * Fout(r, Entree) * Fout(e, Ingredient) +\\
&\quad |Guide| * Fout(Guide, Restaurant) + |Guide| * Fout(Guide, Restaurant) *\\
&\quad |DEL_{prim}| + |Guide| * Fout(Guide, Restaurant) * Fout(r', Name) +\\
&\quad |Guide| * Fout(Guide, Restaurant) * |DEL_{prim}| * Fout(e', Ingredient) +\\
&\quad |DEL_{prim}| * Fout(DEL_{prim}, Name) + |DEL_{prim}| * Fout(DEL_{prim}, Ingredient)
\end{aligned}$$

As in the previous example, we assume that

$$|DEL_{prim}| = |e| * Selectivity(where) = |r| * Fout(r, Entree) * Selectivity(where) \leq 1$$

Note that DEL_{prim} is really constrained not only by $|r|$ and the selectivity of the *where* clause, but also by the selectivity of the nested subquery. That is, the nested subquery further limits the possible removed objects to those that appear in the view because of the old value of &26. We are ignoring the filtering effect of the nested subquery; this is a worst-case assumption for our incremental maintenance techniques, and does not affect the cost of recomputation. □

If the selectivity of the *where* clause of a query is $a\%$, then only $a\%$ of all the objects that satisfy the view specification before applying the *where* clause are actually in the view. In order for an atomic value change from *OldVal* to *NewVal* to be relevant, the truth value of the *where* clause needs to change when *OldVal* is substituted by *NewVal*. As the matrix in Table 7.3 shows, an atomic value change to object o causes insertions to the view $a(100-a)\%$ since $(1-a)\%$ of the time o is not in the view before the change and after the change o is in the view $a\%$ of the time. Similarly, an atomic value change causes deletions to the view $a(1-a)\%$ of the time. When computing the average cost of incremental maintenance after an atomic value change, we multiply the costs of updating the view by $a(1-a)$ to take the relevance of the update into account. In the extreme case, when the selectivity of the *where* clause of the view specification statement is 100% (or 0%), no incremental maintenance statements are required after atomic value changes: they cannot change the set of view objects (of course, the new value of the changed object needs to be installed in the view). For example, if a *where* clause of a view specification contains the clause `x > 0`, and all objects bound to x are greater than 0, then an atomic value change of an object that was bound to x during view materialization from value 2 to value 3 will not cause objects to be added to or removed from the view.

7.8 Evaluation

Our evaluator program accepts a single view specification statement, a database, and a single update, and computes the cost for both recomputation and incremental maintenance using our cost model. (In reality, our evaluator program takes statistics and not an actual database, but we describe the databases themselves here for better presentation.) Here we

		OldVal true	*OldVal* false
NewVal	true	$a \cdot a$	$a(1-a)$
NewVal	false	$a(1-a)$	$(1-a)(1-a)$

Table 7.3: Truth value of the *where* clause for *OldVal* and *NewVal*

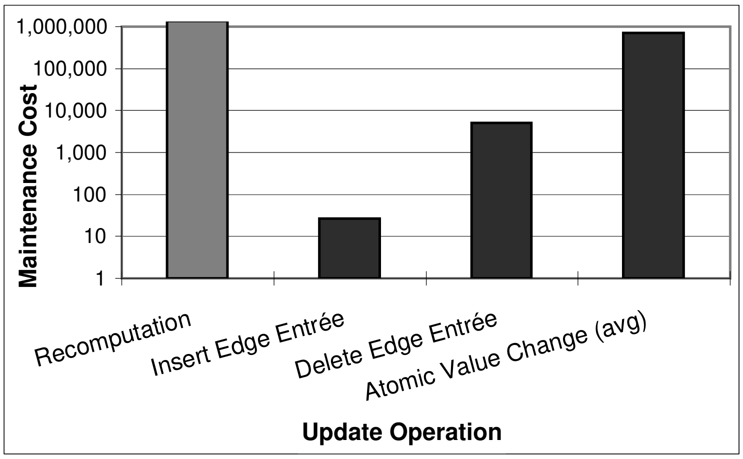

Figure 7.15: Base costs for update operations

present the costs for a variety of view specifications, databases, and updates. We do not use the auxiliary structure *RelevantOids* in the cost model, so the actual costs for incremental maintenance will be lower in many cases. In all of our graphs, the cost is shown on the y axis in a *logarithmic* scale.

Experiment 7.8.1 (Base Costs for Update Operations) In the first experiment, shown in Figure 7.15, we looked at the costs of different update operations for the view specification of Example 7.5.1. The test database was a synthetically generated version of the Guide database containing one Guide, 1000 restaurants, on average 100 entrees and 1 name per restaurant, and 10 ingredients and 2 names per entree. We assumed a fixed selectivity for the *where* clause of 50%. Each bar shows the cost of maintaining the view after a single update for a different update operation.

Recomputation is over 100 times more expensive than incremental maintenance for insert or delete edge operations. These savings are due to binding the variables associated with the inserted or deleted edge. A much smaller portion of the database is traversed during execution of the incremental view maintenance statements compared to the view

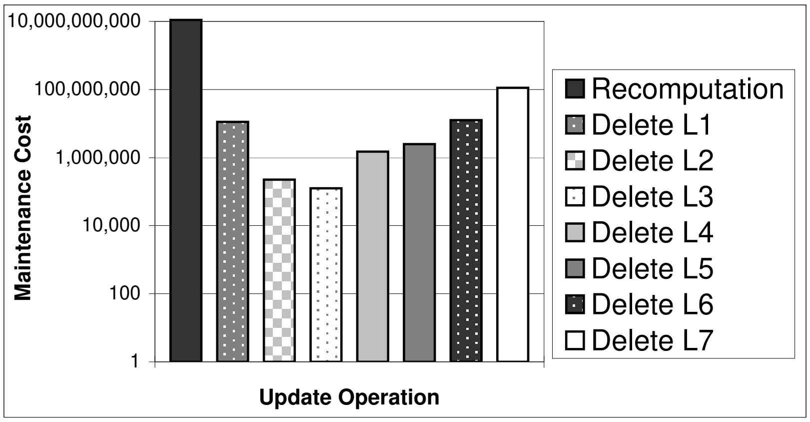

Figure 7.16: Varying position of bound variable in *from* clause

specification statement. Maintaining a view for edge insertions was significantly cheaper than for edge deletions since delete edge maintenance statements require a subquery.

The maintenance cost for an atomic value change can vary significantly. Without the procedure *RelevantVars*, the incremental algorithm will generate a maintenance statement for each condition in the *where* clause. Although each statement will incorporate a variable binding for the changed object, there is only one such binding. Depending on where the binding occurs, the maintenance statement cost may vary from much to only slightly cheaper than the cost of recomputation. Recomputation may be more cost effective depending on the *where* clause. For example, for the view in Example 7.5.1, testing a single atomic value change against both conditions in the *where* clause cost is almost as expensive as recomputation, as shown in Figure 7.15. However, relevance tests using *RelevantOids* can often determine that only a few or even none of the conditions in the *where* clause are relevant. For the same example, evaluating a single maintenance statement is always cheaper than recomputation. □

Experiment 7.8.2 (Bound Variable Position) The position of the bound variable affects the cost of incremental maintenance. For our next experiment, we used a view specification containing a chain of eight path components in the *from* clause:

define view *VaryingFrom* **as** *VF* =
  select z_2 **from** A.L_1 z_1, z_1.L_2 z_2, ..., z_7.L_8 z_8;

The database contained a single named object A, 1000 L_1 subobjects of A, on average 100 L_2 subobjects per z_1, and ten L_i subobjects per z_{i-1} for $3 \leq i \leq 8$. We deleted the

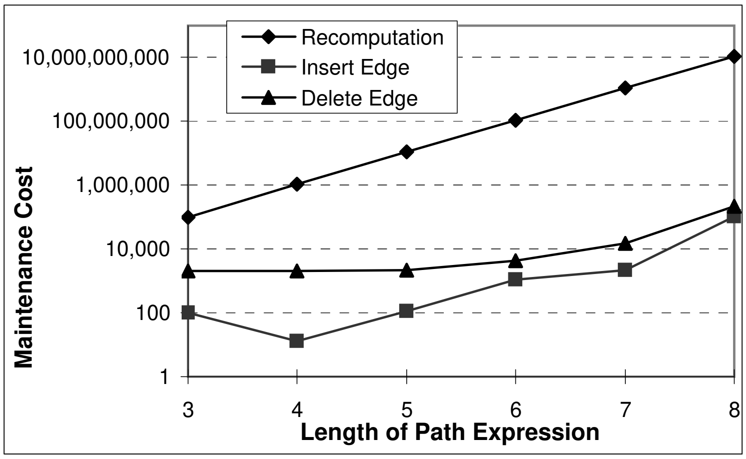

Figure 7.17: Varying length of *from* clause

edge $\langle o_{i-1}, L_i, o_i \rangle$, for all values of $3 \leq i \leq 8$ in turn. Figure 7.16 shows that recomputation is 10–500 times more expensive than incremental maintenance. When the bound variable is in the middle of a path expression, it effectively divides the path into two shorter paths: to compute the total cost, the costs of the two shorter paths need to be added rather than multiplied (see Section 7.7). Therefore, the variable binding provided by the newly inserted or deleted edge has the most beneficial effect when it occurs in the middle of the path expression. □

Experiment 7.8.3 (Length of the *from* Clause) The number of variables in the *from* clause also affects the cost of incremental maintenance. For this experiment, we used view specifications of the following pattern and varied the length of the path expression in the *from* clause from three to eight path components.

define view *VaryingFrom2* **as** VF_2 =
  select z_2 **from** A.L_1 z_1, z_1.L_2 z_2, ..., z_{n-1}.L_n z_n;

The database was the same as in Experiment 7.8.2. For each view specification, we inserted the edge $\langle o_1, L_{\lfloor n/2 \rfloor+1}, o_2 \rangle$, which bound the middle variable in the path. Figure 7.17 shows that as the number of variables increased, the recomputation cost also increased. Each additional edge in the *from* clause caused the relevant portion of the database to increase by a factor of ten. The incremental maintenance costs are much lower and increase more slowly due to the bound variables. The insert edge cost decreases when $n = 4$ because the bound variable appears in the second position for variables z_1 and z_2 for a path expression

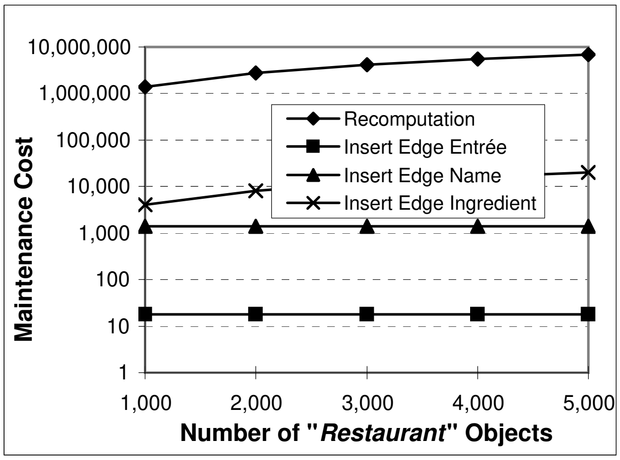

Figure 7.18: Varying database size

of length 4. For this particular database, where there are ten times more objects bound to z_2 than z_3 and z_4, limiting the object bound to z_2 has a larger impact as seen in Figure 7.17. □

Experiment 7.8.4 (Database Size) For the fourth experiment, we used the view specification found in Example 7.5.1, but varied the size of the relevant portion of the database. We increased the number of restaurants in the database from 1000 to 5000, and kept the same average number of entrees per restaurant, and ingredients per entree. Therefore, when the number of restaurants doubled, for example, the size of the relevant portion of the database doubled. The maintenance costs after various edge insertions are shown in Figure 7.18. The cost of recomputation is consistently 100–100,000 times higher than the cost of incrementally maintaining the view.

The size of the database had a negligible effect on inserting an *Entree* and *Name* edge, since the inserted edge provided a binding to a specific restaurant. When inserting an *Ingredient* edge, the placement of the bound variable was not as fortunate because the bindings provided by an *Ingredient* edge insertion did not provide a binding for the variable bound to restaurants. Thus, as the number of restaurants increased so did the cost of finding all *Ingredient* objects. The cost of incremental maintenance for the insertion of an *Ingredient* edge was still many orders of magnitude lower than the cost of recomputation. The recomputation cost always grew linearly with the size of the relevant portion of the database, since it traversed the entire relevant portion.

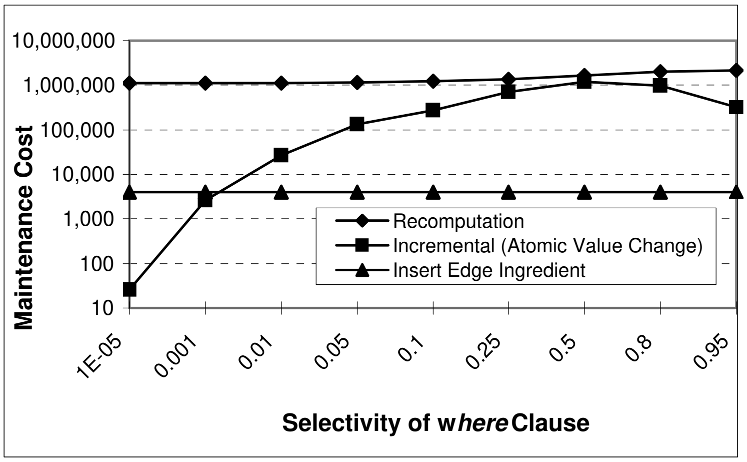

Figure 7.19: Varying selectivity of *where* clause

The graph also shows that the incremental maintenance cost for atomic value changes grew linearly with the database size. This result is unexpected: the change should constrain the query to a small, local portion of the database, regardless of the database's total size. However, the top-down query execution strategy we are using causes this result. Since the *from* clause is always evaluated first, and the specific change only provides a binding in the *where* clause, the same part of the database needs to be traversed in the *from* clause, regardless of the change itself. □

Experiment 7.8.5 (Selectivity of the *where* Clause) Figure 7.19 shows the results of the fifth experiment. We kept the same view definition and database structure as in Example 7.8.1 but varied the selectivity of the *where* clause. As the selectivity increases, more objects are included. Therefore, the recomputation cost went up reflecting the rising cost of locating and adding the adjunct objects. The incremental maintenance cost for atomic value changes is also influenced significantly by the selectivity of the *where* clause. When the selectivity is low, most atomic value changes can be screened out by the syntactic relevance test before running any queries. When the selectivity is high, most objects are already included in the view, so very few new objects need to be added to the view because of the change. Since syntactic relevance tests only apply to atomic value changes (and affect their cost), the maintenance cost for an edge insertion does not change based on the atomic values and the selectivity.

Note that in all our other experiments, the selectivity of the *where* clause is fixed at 50%, which, as shown in Figure 7.19, is the value that most heavily disadvantages our incremental maintenance algorithm. □

Experiment 7.8.6 (Number of Label Occurrences)

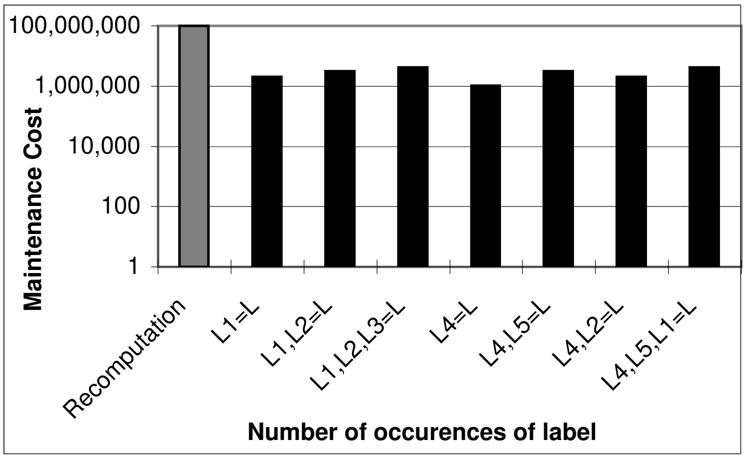

Figure 7.20: Varying number of occurrences of a label in view specification

For the final experiment, we varied the number of times the label of the inserted or deleted edge matched a label in the view specification. We used view specification statements of the following form:

define view *VaryingLabel* **as** $VL =$
  select x
  from A.L_1 x, x.L_2 y, y.L_3 z
  where exists t **in** $y.L_4$: $t < 10$
  and **exists** w **in** $z.L_5$: $w > 7$
  with x.L_6;

We inserted or deleted the edge $\langle o_1,L,o_2\rangle$. For each test, we changed some of the labels in the view specification (as well as the corresponding labels in the source database) to "L", as indicated by the legend for the results, shown in Figure 7.20. The database contained 100 subobjects of each object for each distinct label.

The recomputation cost was unaffected by the specific labels, since the structure of the database remained the same. The incremental maintenance costs varied, however, since each appearance of the label L required an additional maintenance statement. However, even when the label L appeared three times in the view specification, incremental maintenance was still 20 times cheaper than recomputation. □

7.9 Related Work

View mechanisms and algorithms for materialized view maintenance have been studied extensively in the context of the relational model [BLT86, GMS93, GM95, RCK$^+$95, GL95]. Incremental maintenance has been shown to dramatically improve performance for relational views [Han87]. Views are much richer in object-oriented database systems [AB91] and, subsequently, languages for specifying and querying materialized views are significantly more intricate [AB91, Ber91, SAD94, SLT91, Run92].

Previous results on incremental view maintenance for object databases [Run92, RNS96] and nested data [KLMR97] are based on extensive use of type information. Semistructured data provides no type information, so the same techniques do not apply. In particular, subobject sharing along with the absence of a schema makes it difficult to detect if a particular update affects a view. [GGMS97] uses a view maintenance scheme that is limited to a subset of OQL view definition; certain joins are not handled by their algorithms. Most nontrivial semistructured view definitions do not fall within the boundaries of their maintenance algorithm.

[Suc96] also considers incremental view maintenance for semistructured data. The view specification language is limited to select-project queries and only considers database insertions. Our approach allows joins in the view query and handles database insertions, deletions, and updates. [ZG98] investigate graph-structured views and their incremental maintenance. However, their views consist of object collections only, while we include edges (structure) between objects. Also, their maintenance algorithms only work for select-project views over tree-structured databases, while our approach handles joins and arbitrary graph-structured databases.

Chapter 8

External Data Manager

One of the advantages of a DBMS that stores semistructured data, like Lore, is the ability to integrate information easily from heterogeneous information sources without costly transformations. In this chapter we introduce the *external data manager*, a component of Lore that allows for the dynamic integration and caching of external data. The external data manager integrates data stored externally with local data during query processing, and the distinction between local and external data is invisible to the user. Because external data can often be expensive to access, we introduce some optimizations that reduce the amount of data fetched from an external source during the processing of a query, as well as reducing the number of fetches that occur over time.

The material presented in this chapter appeared originally in [MW97].

8.1 Introduction

Lore's external data manager provides a mechanism for dynamically fetching, caching, and querying data stored at any number of heterogeneous sources, and integrating the external data seamlessly with data resident in the Lore system. As an example, consider a database consisting of information about states and regions. While most geographic information is stable, some information, such as weather data, is best obtained dynamically from outside information sources. A *data warehousing* approach [Wid95, LW95] would require external weather data to be integrated into the local database every time it changed, regardless of whether it was needed by a user. In contrast, a fully *on-demand* or *mediated* approach [Wie92] would require that all data, including stable geographic information, be obtained

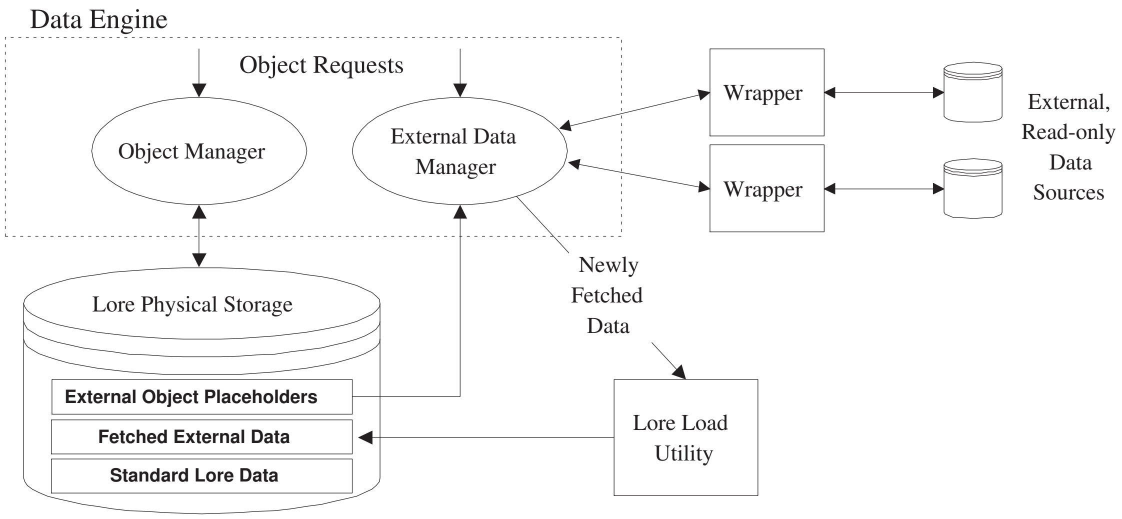

Figure 8.1: The external data manager architecture

from external sources. Our hybrid approach allows the stable information to be stored permanently in Lore, while the dynamic information is fetched (and cached) on demand when needed to answer a user's query.

There are many possible approaches that can be taken to integrate external data into a semistructured DBMS. Our main motivations in choosing the approach described here are to: (i) enable Lore to bring in data from a wide variety of external sources; (ii) make the distinction between local and external data invisible to the user; and (iii) introduce a variety of *argument types* and optimization techniques to limit the amount of data fetched from an external source.

In Section 8.2 we describe the architecture of the external data manager and how it fits in the Lore system. Further details on how we handle external data, especially our methods for reducing calls to external sources and reducing the amount of data retrieved, are described in Section 8.3. Related work is presented in Section 8.4.

8.2 Architecture

In Chapter 2 (Section 2.5, Figure 2.7) we briefly introduced the external data manager in relation to the overall Lore system architecture. In Figure 8.1 we focus in more detail on how the external data manager interacts with Lore's data engine, loader, and external sources. During query processing, requests for objects are sent from the *physical operators* in the data engine and are handled either by the *Object Manager* or the *External Data Manager*. The Object Manager services requests for local data stored under Lore's control. The External Data Manager functions as the integrator between information stored at an external source and the local database—it is responsible for: (i) constructing requests to external sources based upon the current query and the local database state; (ii) caching fetched external data along with information about the requests that generated it; and (iii) seamlessly integrating the external data during query processing.

There are three other components of the Lore system described in Chapter 2 (Section 2.5, Figure 2.7) that may request objects from both the Object Manager and the External Data Manager. The *statistics manager* and *index manager* will visit objects in the database during the creation of their secondary structures. Neither component gathers information about data stored at an external source, so external objects and their subobjects are skipped. The *API* component supports arbitrary graph traversal for applications. When using the API for graph exploration, there is no notion of query or current path. Therefore, the external data manager will not issue any requests to an external source and will present only cached external data to the API.

The *wrapper* modules shown in Figure 8.1 accept requests from the external data manager (implemented as calls to the wrapper program) and translate them into specific commands for the external source. They also translate results from the external source into OEM before returning them to the external data manager.

The data stored within a Lore database can be divided into three categories: standard data, *External Object Placeholders*, and *Fetched External Data*. An external object placeholder, which is invisible to the user, specifies how Lore interacts with an external data source. Fetched external data consists of objects cached within Lore as a result of calls to external data sources. These objects are queried and retrieved just like standard Lore objects. In a Lore database, the external object placeholder and fetched external data for a single external source are stored as subobjects of a single object that we refer to as an *external object*. Details of the representation can be seen in Figure 8.2 and are described later. The *Lore Load Utility* is a bulk loader designed to quickly load large amounts of data. It is shown in the figure because it is used by the External Data Manager to load into Lore the data returned from calls to external sources.

As a concrete example of how the components interact, consider the sample OEM database shown in Figure 8.2, ignoring for now the structure of the shaded (external) object and its subobjects. Suppose the query execution engine requests the `Name` subobject of the `State` object. Since the requested object is standard Lore data, the Object Manager handles the request. However, if the `Weather` subobject of `State` is requested, the request is handled by the External Data Manager, which may send a sequence of requests for information to the external source (further discussed in Section 8.3). Each external request is logged and the results are stored in the database via the load utility. After all requests are complete, the External Data Manager provides back to the query execution engine a set of objects corresponding to the relevant external information.

It is important to note that the external placeholder data, which in Figure 8.2 consists of all subobjects of the external object except portions of the one labeled as "`Fetched_Data`", is used by the external data manager only and is not visible to the query execution engine or the user. External object placeholders are created by database administrators when a Lore database is created.

8.2.1 Limitations

We impose two restrictions on how the external data manager is used in the Lore system. First, we require the query engine to use a top-down query execution strategy (Chapter 3, Section 3.3) for all queries that could potentially encounter an external object. We have not considered fine-grained methods to determine when a query may encounter an external object. Currently, Lore simply sets a flag whenever a database contains an external object, and forces top-down query plans when the flag is set. As a second restriction, all queries over databases that contain external objects must be expressed in disjunctive normal form (DNF) as defined in Chapter 2 (Section 2.4.5).

8.3 Details

Figure 8.2 illustrates a tiny portion of a geographic/weather database, as motivated in Section 8.1. The data includes some Lore-resident data about states and regions, along with a single external object (shaded in the figure) that fetches up-to-date weather information from an external source based on a geographic area such as a state or region, or (as will be seen) based on a city within an area. The illustrated database includes information about a single state, "California", and a single region within the country "USA" which may be referred to either as "New England" or "Northeast".

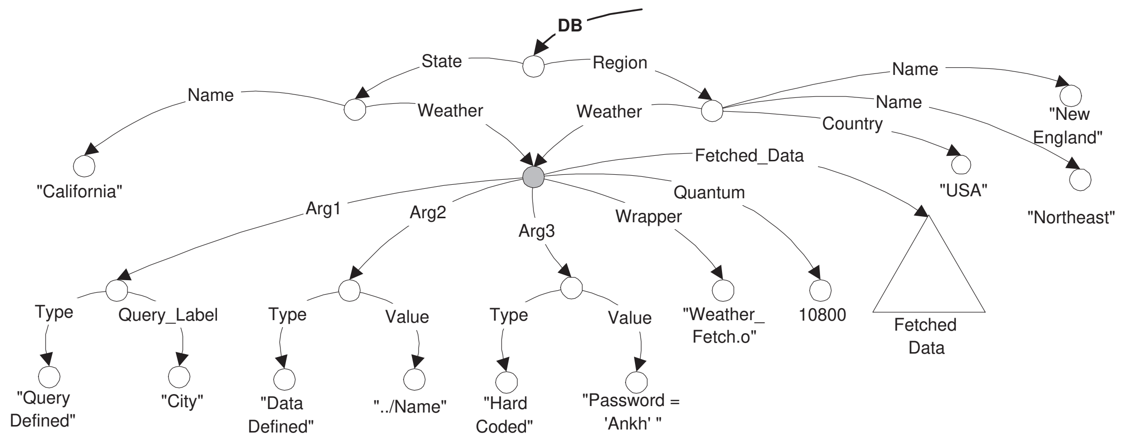

Figure 8.2: An example OEM database with an external object

As mentioned in Section 8.2, all subobjects of the shaded object except `Fetched_Data` constitute the external object placeholder. The `Quantum` subobject indicates the time interval (in seconds) until cached external information becomes "stale" and is no longer used. The `Wrapper` subobject specifies the program that interfaces between Lore and the external source. `Arguments` sent to the wrapper program (`Arg1`, `Arg2`, and `Arg3` in the figure) provide a way to qualify the information requested. Arguments limit the data fetched from an external source to that which is immediately useful in answering the current query, thus: (i) reducing the cost associated with shipping data from an external location; (ii) reducing the amount of external data stored within Lore; and (iii) speeding up query processing by reducing the number of objects examined. Arguments sent to the external source can come from three places: the query being processed (*query-defined arguments*), values of Lore-resident objects (*data-defined arguments*), and constant values tied to the external object (*hard-coded arguments*).

8.3.1 Single Argument Values

First, we consider how single argument values are extracted from the user's query (query-defined arguments), from the database state (data-defined arguments), and from the external object placeholder (hard-coded arguments). In Section 8.3.2 we will show how arguments are combined into actual calls to the external source. The values of query-defined and data-defined arguments rely on the concept of *current database path* to the external object. The current path taken to discover an external object can be extracted from the current *evaluation* (recall Chapter 3, Section 2.5.1) being operated on by the query execution engine. Recall from Section 8.2.1 that only top-down execution strategies are being considered. For Figure 8.2 there are two possible current paths to the single external object: the one with labels "`DB.State.Weather`" and the one with labels "`DB.Region.Weather`".

We first describe a somewhat simplified approach for generating arguments using the database shown in Figure 8.2. We then explain the full generality of our approach. Hard-coded arguments, such as `Arg3`, supply their value directly from within the external object placeholder. A query-defined argument, such as `Arg1`, always includes a `Query_Label` subobject in its placeholder. The labels making up the current path, followed by the specified `Query_Label`, say L, make up a path expression P. If the *where* clause of the current query contains $P = v$ for some constant value v, then $L = v$ becomes a value for the query-defined argument. A data-defined argument can either point directly to an atomic object in the database whose value will be sent as an argument (not illustrated in Figure 8.2), or it can specify objects via a "relative path" through the database as illustrated by `Arg2`. The relative path is evaluated with respect to the current path, resulting in zero or more objects whose values become arguments. Note that ".." in the relative path means to traverse up to the parent object in the current path, in the Unix style.

To be more concrete, suppose in Figure 8.2 the query processor calls the external data manager for the shaded external object via the current path labeled "`DB.Region.Weather`". Consider `Arg1`. The `Query_Label` subobject with value "City" specifies that if the query being processed has any predicate of the form "`DB.Region.Weather.City = X`" in its *where* clause, then "$City = X$" is an argument value. In `Arg2`, the `Value` subobject specifies the relative path expression `"../Name"`. Based on the current path, the possible data-defined argument values are "*Name = 'New England'*" and "*Name = 'Northeast'*". Finally, `Arg3` is a hard-coded argument specifying "*Password = 'Ankh'*".

As mentioned above, our description of argument generation has so far been somewhat simplified. In the general case, each argument descriptor in the external object placeholder may include an optional `Tag` and `Operator` subobject, with atomic values t and `op` respectively. If these subobjects are included, once we have obtained an argument value v as described above, the actual argument sent to the source is "t *op* v". (Thus, in the simplified examples above, we have assumed tags `City`, `Name`, and `Password`, and operator =, although they are not shown in the figure.) In query-defined arguments, the relevant operator in the query must match the operator in the argument descriptor. However, as a further enhancement, a special operator `match` is permitted for query-defined arguments, which causes the corresponding operator in the query to be sent as part of the argument: In our example, if the query included "`DB.Region.Weather.City like` *X*" instead of "`DB.Region.Weather.City = X`", then the argument sent to the external source would be "*City like X*". This feature allows queries to easily exploit operators available in external sources. Finally, if no `Tag` or `Operator` is specified, then the argument is sent without qualification.

8.3.2 Argument Sets and Calls to External Source

A single object request during the execution of a query to the external data manager may result in multiple calls to the external source due to multiple bindings for data-defined and/or query-defined arguments. A sequence of argument sets is generated, and each argument set results in (at most) one call to the external source. Pseudocode to generate the sequence of argument sets appears in function *CreateArgumentSets* shown in Figure 8.3. *CreateArgumentSets* accepts as input an external object and a set of disjuncts derived from the query's *where* clause. Recall from Section 8.2.1 that the query must by expressed in disjunctive normal form. We eliminate those disjuncts that do not reference the variable that was bound to the external object. *CreateArgumentSets* returns as output a list of strings, where each string specifies an argument set for a separate request to the external source.

We now describe *CreateArgumentSets* in more detail. In line 2 the algorithm extracts the hard-coded arguments, since these do not change and will be included in all argument sets. Line 3 begins a loop that will iterate over each disjunct in the query. Each disjunct can create many argument sets depending on the number of data-defined arguments and the number of matching data-defined values in the database. In line 4 the query-defined arguments for the current iteration are extracted as described in Section 8.3.1. These arguments will be included in every argument set generated by lines 5 through 23, which handle data-defined arguments. If there are no data-defined arguments (the `if` on line 5) then the argument set consists solely of the hard-coded and query-defined arguments. Otherwise, line 8 begins a loop that packages the hard-coded, query-defined, and all possible combinations of data-defined values into a single argument set, which is added to the result in line 14. The data-defined values are extracted by a sequence of GetFirstValue (line 9)

```
function CreateArgumentSets(ExternalObject EO, Disjuncts disjuncts)⇒ List⟨String⟩
    List⟨String⟩ result = NULL;
    String HCA = HardCodedArgs(EO.HardCodedArgs);
    foreach D in disjuncts do
        String QDA = QueryDefinedArgs(EO.QueryDefinedArgs, D);
        // Check for existence of data-defined arguments
        if (# DataDefined Args in EO = 0) then
            // Concatenate strings and add to result
            result += PackageArguments(HCA, QDA, NULL);
        else
            // Reset all data-defined arguments to their first values
            For I = 1 to # DataDefined Args in EO
                DDA[I] = GetFirstValue(EO.DataDefinedArgs, i);
            // Do a pairwise join between all possible argument values
            curDDA = 0;
            while (curDDA <= # DataDefined Args)
                foreach V in the first DataDefined Arg
                    DDA[1] = V;
                    // Add arguments to the result
                    result += PackageArguments(HCA, QDA, DDA);
                // Determine next combination of data-defined values
                curDDA = 2;
                while ((the Current value for DDA[curDDA] is the last value) and
                   (curDDA <= # DataDefined Args))
                    ResetToFirstValue(EO.DataDefinedArgs, curDDA)
                    DDA[curDDA] = GetFirstValue(EO.DataDefinedArgs, curDDA);
                    curDDA++;
                // End case for is when all data-defined combinations have
                // been considered
                if (curDDA <= # DataDefined Args) then
                    DDA[curDDA] = GetNextValue(EO.DataDefinedArgs, curDDA);
    return result;
```

Figure 8.3: Pseudocode to generate all argument sets

and GetNextValue (line 23) calls, which simulate a general counting mechanism that builds argument sets that are a combination of all possible values.

We illustrate the use of *CreateArgumentSets* with the following example.

Example 8.3.1 Suppose the following query is issued over the database shown in Figure 8.2.

```
select w
from DB.Region.Weather w
where w.City = "Providence"
```

The variable w is bound to the single external object and *CreateArgumentSets* is called with a single disjunct containing the predicate "`w.City = 'Providence'`". The single hard-coded argument extracted from the external object in Figure 8.2 is "Password = 'Ankh'", so it is assigned to *HCA* in line 2 of *CreateArgumentSets*. Query-defined argument "City = 'Providence'" is assigned to QDA in line 4. There is one data-defined argument with a relative path that binds it to two separate values: "New England" and "Northeast". The first value is assigned to DDA[1] in line 13. The call to *PackageArguments* in line 14 concatenates the hard-coded, query-defined, and data-defined arguments into a single string and adds that string to the result. On the second pass through the line 12 `for` loop we have the same behavior with DDA[1] = "Northeast". When line 16 is reached *result* contains the two argument sets: "Password = 'Ankh', City = 'Providence', Name = 'New England'" and "Password = 'Ankh', City = 'Providence', Name = 'Northeast'". The `while` loop in line 17 and the `if` statement on line 22 are not entered since there is only one data-defined argument. The `while` loop of line 11 ends and the function returns the *result* shown above. □

After the argument sets have been computed, information is obtained from the external source by making at most one call for each argument set. Fewer calls can be made based on using cached data as described in the next paragraph and/or based on optimizations described in Section 8.3.3.

All information fetched from the external source is stored under the `Fetched_Data` edge of the external object (recall Figure 8.2). Stored with each batch of fetched data, but invisible to the user, is the argument set that was sent to the source and resulted in the fetched data, along with the time when the data was fetched. Before each call is made to the external source, the external data manager first checks to see if the exact same argument set (or a subset of the argument set as explained in Section 8.3.3) has already been sent to the external source. If so, and if the associated fetched data has not expired (based on its fetch time and `Quantum` subobjects, recall Figure 8.2), then no call is made to the external source and the cached data is used. If the call has previously been made but the data is stale then we refetch the data from the external source. Finally, if the argument set has never been seen before then a call to the external source is made and the fetched data is cached.

8.3.3 Optimizations

Many calls to an external source can quickly dominate query processing time, so we incorporate certain optimizations to limit the number of calls to an external source. We make the assumption that more arguments result in less fetched data.[1] First, we order the sequence of calls so that less restrictive argument sets are sent to the external source first. This is done by ordering the query disjuncts passed into *CreateArgumentSets* (Figure 8.3) so that those disjuncts that provide more query-defined values will come later in the generated sequence of argument sets, i.e., the disjuncts are sorted by increasing number of query-defined arguments for the current path. We thus increase the odds that a given fetch will *subsume* a later fetch in the sequence: Argument sets used previously (by the current query or an earlier one) are tracked as described in Section 8.3.2. Previously fetched (non-expired) information is guaranteed to include all data that would be fetched by the current argument set if the current argument set is a superset of the previous one. When subsumption occurs, the subsumed argument set is discarded.

Example 8.3.2 We will illustrate the subsumption optimization by showing a sequence of three queries. The queries are posed over the database in Figure 8.2, and we assume that these are the first queries issued to that database. To begin, we issue the following query, which asks for all state-based weather:

```
select w
from DB.State.Weather w
```

[1] Although this property may not seem obvious at first, most external sources do behave in this manner. Consider, for example, a relational database with selection conditions, or a web site with an attribute-value forms interface.

This query generates a single argument set with one data-defined argument and one hard-coded argument: “Name = ’California’, Password = ’Ankh’”. This argument set is sent to the external source, and the data returned from the source is cached along with the argument set and the time that it was fetched.

The second query asks for the weather in cities named “Smithville”:

```
select w
from DB.State.Weather w
where w.City = "Smithville"
```

This query generates a single argument set with one argument of each type: “City = ’Smithville’, Name = ’California’, Password = ’Ankh’”. This argument set is a subset of the previous argument set. Therefore, if the cached data from the previous query has not expired, then that data can be used and no external call is sent. If the cached data has expired then the argument set generated by this query is sent to the external source.

The third query asks for regional weather information for the city “Boston” and for any region where the low temperature is below 40.

```
select w
from DB.Region.Weather w
where w.City = "Boston" or
      w.LowTemperature < 40
```

The four argument sets generated for this query are shown (in the order that the sets are generated by `CreateArgumentSets`) in Figure 8.4. In generating the four argument sets the algorithm considers the second disjunct first, since it provides no query-defined arguments for the current path. The data-defined argument is evaluated with respect to the current path “`DB.Region.Weather`” yielding two data defined values and the first two argument sets in Figure 8.4. The last two argument sets are generated by the first disjunct and include the query defined argument `City="Boston"`. The data fetched by argument sets 3 and 4 will be subsumed by the data fetched by argument sets 1 and 2. Therefore, only two fetches will be made to the external source. □

1	Name = “Northeast”, Password = “Ankh”
2	Name = “New England”, Password = “Ankh”
3	Name = “Northeast”, City = “Boston”, Password = “Ankh”
4	Name = “New England”, City = “Boston”, Password = “Ankh”

Figure 8.4: Argument sets generated by the external data manager

8.4 Related Work

There has been a huge amount of research devoted to the general topic of data integration, e.g., [BLN86, LMR90b, HM93, LRO96, PGMW95, PGH96]. Most of this work considers a *mediated* environment where heterogeneous sources are connected and their data is integrated via query processing middleware. At the other extreme, *data warehouses* integrate all data in advance of any query processing [Wid95, LW95]. As far as we are aware, ours is the only work to consider the dynamic integration of data from external, heterogeneous sources during query processing in the context of a DBMS for semistructured data.

Chapter 9

Conclusions and Future Work

This thesis investigates several aspects of data management and query processing for semistructured data.

Our work is based around the *Lore* database management system for semistructured data and its query language *Lorel* (Chapter 2). While the overall architecture of Lore is similar to relational and object-oriented DBMSs, each of the components needed significant modifications to deal with the semistructured nature of the data managed by Lore.

We presented a cost-based query optimization framework for Lore (Chapter 3). The framework constructs flexible logical query plans that can easily be transformed into a wide variety of physical query plans. We discussed a search strategy and pruning heuristics for creating physical query plans that are appropriate for semistructured data. We identified the database statistics that are required to optimize Lorel queries.

We investigated several specialized query optimization techniques for semistructured data (Chapters 4, 5, 6). These techniques focus on particular constructs in the query language or particular database "shapes". They consist of query rewrites (applied before construction of a physical query plan), optimizations during physical query plan generation, and post-optimizations (applied after constructing a physical query plan). We have included performance analyses for each of the techniques.

We introduced a view definition and management facility for semistructured data (Chapter 7). We developed algorithms to incrementally maintain materialized views, and analyzed the performance of our incremental algorithms as compared to full recomputation. We concluded that our incremental maintenance algorithms are preferred in a vast majority of situations.

We described the *external data manager* we built in Lore (Chapter 8). This component allows Lore to dynamically integrate data from external sources into a local Lore database during query processing in a manner that is invisible to the user.

9.1 Future Work

We now describe several potential areas for future work.

9.1.1 Physical Parent Pointers

As described in Chapter 2 (Section 2.5.2), Lore can build and maintain a *Lindex*, which supports finding all parents of a given object reachable via a given label. Instead of maintaining a separate index, we could instead augment our storage manager to store directly with objects their parent pointers in addition to their subobject (child) pointers. While parent information would then be readily available when an object is in memory (and not require an additional index probe), this information would increase the size of memory required by every object and thus could result in more overall disk activity. An analysis between the two alternatives needs to be done to determine the overall best approach.

9.1.2 Statistics and Object Placement

The cost model used by Lore's query optimizer is based on the number of object requests rather than the more accurate measure of page requests used by most commercial DBMSs. Lore's current architecture does not allow for specific placement of objects on pages, and we do not gather page-level statistics, so it is impossible to predict page requests to any level of accuracy. This simplistic cost model can potentially lead to poor choices by the query optimizer. We could introduce additional statistics by gathering information about the clustering of objects on pages. Alternatively, we could introduce object placement policies that would attempt to cluster objects together on pages when they are likely to be fetched together. While such schemes are usually only heuristics, huge benefits could be gained by good object placement.

9.1.3 Further Optimizations for Path Expressions

Although we have considered a wide variety of algorithms to optimize path expressions in Chapter 6, combinations of our optimization techniques could also be considered. For example, the *Branches* algorithm (Chapter 6, Section 6.2.2) could call any of the other algorithms in Chapter 6 in order to optimize individual branches. It might also be interesting to compare the algorithms in Chapter 6 against more traditional query optimization search strategies, such as exhaustive bottom-up (system R style) [OL90, PGLK97, SAC+79], transformation-based search using iterative improvement or simulated annealing [IK90, Swa89], and random search [GLPK94]. Finally, since this work has focused on optimizing path expressions in isolation, there is further work to be done to generalize the techniques for complete Lorel queries.

9.1.4 Further Work on Compile-Time Path Expansion

In Chapter 4 (Section 4.2) we introduced query rewrite techniques to remove regular expression operators from general path expressions. Further work to be done includes:

- Predicting when it is beneficial to expand a general path expression. We need some mechanism to estimate the time to rewrite a general path expression and execute the rewritten query against the time to execute the original query.

- Predicting when it is beneficial to break a path expression containing a `union`. We can feed both choices into the physical query plan generator and cost each (recall Chapter 4, Section 4.2.2). However, this approach can be inefficient since optimization times is non-negligible, so alternative mechanisms to make this decision need to be explored.

9.1.5 Further Work on Incremental View Maintenance

Several optimizations to our incremental maintenance algorithm presented in Chapter 7 (Section 7.3) are possible. First, the algorithm could be extended to handle sets of updates together. Second, if the data has a tree structure, then the maintenance statements can be simplified, e.g., by eliminating the subqueries when deleting objects or edges. Third, we would like to incorporate query optimization and query rewriting techniques from Chapters 3 and 4, and provide more query execution choices to the query optimizer.

9.1.6 Extensions to the External Data Manager

The external data manager (Chapter 8) dynamically fetches and caches external data during query processing. There are a number of extensions that could be made to the this system:

- Cached external data becomes stale based on a simple timeout mechanism. For those external sources with a triggering mechanism [WC96], cached data could become stale based on notifications from the source to the external data manager that the data has changed.
- Our current optimization techniques eliminate calls to external sources based on complete subsumption: a call is eliminated if it is guaranteed to return strictly less information than a previous one (Chapter 8, Section 8.3.3). A more general mechanism could share information fetched in multiple calls when the information is known to intersect, but may not satisfy complete subsumption.
- Lore contains a sophisticated index manager, but currently it indexes Lore-resident data only. There are many interesting issues related to indexing external data. For example, how do we maintain an index when the (external) data is updated without notification? Alternatively, an index could be created "incrementally" as data is fetched from the source, but such an index is not guaranteed to be complete or current.

9.1.7 Triggers for Semistructured Data

One fairly standard database feature we have not considered is *triggers* [WC96]. Simple triggers, defined over changes to atomic values or the creation or deletion of edges, could resemble triggers in traditional DBMSs. More complex triggers, such as triggers based on changes to objects reachable via a path, would require a more complex mechanism. A trigger system could use some of the results obtained from the incremental maintenance of materialized views (Chapter 7), since in many cases recognizing when a change affects a view is similar to recognizing when a trigger should be fired.

Appendix A

Lorel Syntax

The complete Lorel syntax appears below. In the grammar "{}*" means 0 or more repetitions, "{}+" means 1 or more repetitions, and "[ ]" means optional. The exception is Rule 21, where [ ] is used to delimit a character class and the following + means that a sequence of one or more characters can be drawn from the class. Rule 19 has higher precedence than Rule 17, meaning that a path expression consisting of multiple label expressions separated by dots is parsed as multiple qualified paths, rather than a single qualified path consisting of multiple paths.

(1) *query*	::=	*set_query*
		| *atomic_query*
		| *value_query*
		| *update_query*
(2) *set_query*	::=	*sfw_query*
		| *path_expr*
		| *set_query* `intersect` *set_query*
		| *set_query* `union` *set_query*
		| *set_query* `except` *set_query*
		| (*set_query*)
(3) *atomic_query*	::=	*var*
		| `element`(*set_query*)

(4) *value_query*	::=	`*`*atomic_query*
		| *constant*
		| `pathof`(*path_var*)
		| *external_pred_or_func*(*query_list*)
		| (*query*) *arith_op* (*query*)
		| − *query*
		| `abs`(*query*)
		| *aggr_function*(*set_query*)
(5) *query_list*	::=	*query*
		| (*query*){, (*query*)}*
(6) *sfw_query*	::=	`select` [`distinct`] *select_expr* {, *select_expr* }*
		[`from` *from_expr* {, *from_expr* }*]
		[`where` *predicate*]
(7) *select_expr*	::=	*query* [`as` *select_identifier*]
		| *select_identifier* : *query*
		| `oem`(*select_expr* {, *select_expr* }*) [`as` *select_identifier*]
(8) *select_identifier*	::=	identifier
		| `unquote`(*path_var*)
(9) *from_expr*	::=	*path_expr* [[`as`] *var*]
		| *var* `in` *path_expr*
(10) *predicate*	::=	`not` *predicate*
		| *predicate* `and` *predicate*
		| *predicate* `or` *predicate*

| *query comp_op query*
| *set_query*
| `exists`(*set_query*)
| *boolean_constant*
| `exists` *var* `in` *set_query* : *predicate*
| `for all` *var* `in` *set_query* : *predicate*
| *query* `in` *set_query*
| *query comp_op quantifier set_query*
| *external_pred_or_func*(*query_list*)
| (*predicate*)

(11) *arith_op* ::= + | − | ∗ | / | `mod`

(12) *comp_op* ::= < | <= | = | <> | >= | >
| `like` | `grep` | `soundex`

(13) *aggr_function* ::= `min` | `max` | `count` | `sum` | `avg`

(14) *quantifer* ::= `some` | `any` | `all`

(15) *constant* ::= `nil`
| integer_literal
| real_literal
| quoted_string_literal
| *boolean_constant*

(16) *boolean_constant* ::= `true` | `false`

(17) *path_expr* ::= *var* {*qualified_gpe_component*}+

(18) *qualified_gpe_component* ::= *gpe_component* [@*path_var*] [{*var*}]

(19) *gpe_component*	::=	. *label_expr*
		| *gpe_component* **|** *gpe_component*
		| *gpe_component* *gpe_component*
		| (*gpe_component*) [*regexp_op*]
(20) *regexp_op*	::=	* | + | ?
	::=	#
(21) *label_expr*		| [A-Za-z0-9%_]+
		| `unquote`(*path_var*)
(22) *path_var*	::=	identifier
(23) *var*	::=	identifier
(24) *external_pred_or_func*	::=	identifier
(25) *update_query*	::=	*value_update*
		| *edge_update*
		| *name_update*
(26) *value_update*	::=	`update` *variable update_op query*
		[`from` *from_expr*]
		[`where` *where_expr*]
(27) *edge_update*	::=	`update` *variable.label_expr update_op query*
		[`from` *from_expr*]
		[`where` *where_expr*]
(28) *name_update*	::=	[`name`] *name_list* := *query*
		| [`name`] *name_list* := `oemnil`

(29) *name_list* ::= *identifier* {, *name_list* }*
| *identifier*

(30) *update_op* ::= =
| +=
| -=

Bibliography

[AB91] Serge Abiteboul and Anthony Bonner. Objects and Views. In *Proc. SIGMOD*, pages 238–247, Denver, Colorado, May 1991.

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

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

[ALW99] S. Abiteboul, T. Lahiri, and J. Widom. Ozone. Working Document, Stanford University Database Group, September 1999.

[AMR+98] S. Abiteboul, J. McHugh, M. Rys, V. Vassalos, and J. Wiener. Incremental maintenance for materialized views over semistructured data. In *Proceedings of Twenty-Fourth International Conference on Very Large Data Bases*, pages 38–49, New York, New York, August 1998.

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

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

[Ber91] Elisa Bertino. A View Mechanism for Object-Oriented Databases. In *Proc. EDBT*, pages 136–151, Vienna, March 1991.

[BF97] E. Bertino and P. Foscoli. On modeling cost functions for object-oriented databases. *IEEE Transactions on Knowledge and Data Engineering*, 9(3):500–508, May 1997.

[BLN86] C. Batini, M. Lenzerini, and S. Navathe. A comparative analysis of methodologies for database schema integration. *ACM Computing Surveys*, 18:323–364, 1986.

[BLT86] José A. Blakeley, Per-Åke Larson, and Frank Wm. Tompa. Efficiently Updating Materialized Views. In *Proc. SIGMOD*, pages 61–71, Washington, D.C., May 1986.

[BPSM98] Editors: T. Bray, J. Paoli, and C. Sperberg-McQueen. Extensible markup language (XML) 1.0, February 1998. W3C Recommendation available at http://www.w3.org/TR/1998/REC-xml-19980210.

[BRG88] E. Bertino, F. Rabitti, and S. Gibbs. Query processing in a multimedia document system. *ACM Transactions on Office Information Systems*, 6(1):1–41, January 1988.

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

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

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

[CCM96] V. Christophides, S. Cluet, and G. Moerkotte. Evaluating queries with generalized path expressions. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 413–422, Montreal, Canada, June 1996.

[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.

[CD92] S. Cluet and C. Delobel. A general framework optimization in object-oriented queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 383–392, San Diego, California, June 1992.

[Com79] D. Comer. The ubiquitous b-tree. *ACM Computing Surveys*, 11:121–137, 1979.

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

[CZ96] M. Cherniack and S. Zdonik. Rule languagnes and internal algebras for rule-based optimizers. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 401–412, Quebec, Canada, June 1996.

[CZ98] M. Cherniack and S. Zdonik. Changing the rules: Transformations for rule-based optimizers. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 61–72, Seattle, Washington, June 1998.

[DFF+99] 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.

[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 web-site management system. *SIGMOD Record*, 26(3):4–11, September 1997.

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

[FNPS79] R. Fagin, J. Nievergelt, N. Pippenger, and H. Strong. Extendible hashing – A fast access method for dynamic files. *ACM Transactions on Database Systems*, 4(3):315–344, September 1979.

[FS98] M. Fernandez and D. Suciu. Optimizing regular path expressions using graph schemas. In *Proceedings of the Fourteenth International Conference on Data Engineering*, pages 14–23, Orlando, Florida, February 1998.

[GGMR97] J. Grant, J. Gryz, J. Minker, and L. Raschid. Semantic query optimization for object databases. In *Proceedings of the Thirteenth International Conference on Data Engineering*, pages 444–454, Birmingham, UK, April 1997.

[GGMS97] Dieter Gluche, Torsten Grust, Christof Mainberger, and Marc H. Scholl. Incremental Updates for Materialized OQL Views. In *Proc. DOOD*, pages 52–66, Montreux, Switzerland, December 1997.

[GGT95] G. Gardarin, J. Gruser, and Z. Tang. A cost model for clustered object-oriented databases. In *Proceedings of the Twenty-First International Conference on Very Large Data Bases*, pages 323–334, Zurich, Switzerland, September 1995.

[GGT96] G. Gardarin, J. Gruser, and Z. Tang. Cost-based selection of path expression processing algorithms in object-oriented databases. In *Proceedings of the Twenty-Second International Conference on Very Large Data Bases*, pages 390–401, Bombay, India, 1996.

[GL95] Timothy Griffin and Leonid Libkin. Incremental Maintenance of Views with Duplicates. In *Proc. SIGMOD*, pages 328–339, San Jose, California, May 1995.

[GLPK94] C. Galindo-Legaria, A. Pellenkoft, and M. Kersten. Fast, randomized join-order selection – why use transformations? In *Proceedings of the Twentieth International Conference on Very Large Data Bases*, pages 85–95, Santiago, Chile, September 1994.

[GM95] Ashish Gupta and Inderpal Singh Mumick. Maintenance of Materialized Views: Problems, Techniques, and Applications. *Bulletin of the TCDE*, 18(2):3–18, June 1995.

[GMS93] Ashish Gupta, Inderpal Singh Mumick, and V.S. Subrahmanian. Maintaining Views Incrementally. In *Proc. SIGMOD*, pages 157–166, Washington, D.C., May 1993.

[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.

[GR90] C. F. Goldfarb and Y. Rubinsky. *The SGML Handbook*. Clarendon Press, Oxford, UK, 1990.

[GR92] J. Grey and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Francisco, California, 1992.

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

[GW97] R. Goldman and J. Widom. DataGuides: Enabling query formulation and optimization in semistructured databases. In *Proceedings of the Twenty-Third International Conference on Very Large Data Bases*, pages 436–445, Athens, Greece, August 1997.

[Han87] Eric N. Hanson. A Performance Analysis of View Materialisation Strategies. In *Proc. SIGMOD*, pages 440–453, San Francisco, CA, 1987.

[HM93] J. Hammer and D. McLeod. Querying heterogeneous information sources using source descriptions. *International Journal of Intelligent and Cooperative Information Systems*, 2:51–83, 1993.

[IK90] Y. Ioannidis and Y. Kang. Randomized algorithms for optimizing large join queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 312–321, Atlantic City, New Jersey, May 1990.

[Imm87] N. Immerman. Languages that capture complexity classes. *SIAM Journal of Computing*, 16(4):760–778, August 1987.

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

[KLMR97] Akira Kawaguchi, Daniel F. Lieuwen, Inderpahl S. Mumick, and Kenneth A. Ross. Implementing Incremental View Maintenance in Nested Data Models. In *Proc. DBPL*, 1997.

[KMP93] A. Kemper, G. Moerkotte, and K. Peithner. A blackboard architecture for query optimization in object bases. In *Proceedings of the Nineteenth International Conference on Very Large Data Bases*, pages 543–554, Dublin, Ireland, August 1993.

[KS91] H. Korth and A. Silberschatz. *Database System Concepts*. McGraw-Hill, New York, New York, 1991.

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

[LMR90a] W. Litwin, L. Mark, and N. Roussopoulos. Interoperability of multiple autonomous databases. *ACM Computing Surveys*, 22(3):267–293, 1990.

[LMR90b] W. Litwin, L. Mark, and N. Roussopoulos. Interoperability of multiple autonomous databases. *ACM Computing Surveys*, 22:267–293, 1990.

[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 Data Bases*, pages 251–262, Bombay, India, September 1996.

[LW95] D. Lomet and J. Widom, editors. *Special Issue on Materialized Views and Data Warehousing, IEEE Data Engineering Bulletin,* 18(2), June 1995.

[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.

[Man98] Udi Manber. Glimpse, February 1998. Located at http://glimpse.cs.arizona.edu/.

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

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

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

[ODE95] C. Ozkan, A. Dogac, and C. Evrendilek. A heuristic approach for optimization of path expressions. In *Proceedings of the International Conference on Database and Expert Systems Applications*, pages 522–534, London, United Kingdom, September 1995.

[OL90] K. Ono and G. Lohman. Measuring the complexity of join enumeration in query optimization. In *Proceedings of the Sixteenth International Conference on Very Large Data Bases*, pages 314–325, Brisbane, Australia, August 1990.

[OMS95] M. T. Ozsu, A. Munoz, and D. Szafron. An extensible query optimizer for an objectbase management system. In *Proceedings of the Fourth International Conference on Information and Knowledge Management*, pages 188–196, Baltimore, Maryland, November 1995.

[O'N87] Patrick O'Neil. Model 204 architecture and performance. In *Proceedings of the 2nd International Workshop on High Performance Transaction Systems (HPTS)*, pages 40–59, Asilomar, CA, 1987.

[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.

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

[PGH96] Y. Papakonstantinou, A. Gupta, and L. Haas. Capabilities-based query rewriting in mediator systems. In *Proceedings of the Conference on Parallel and Distributed Information Systems*, pages 170–181, Miami Beach, Florida, December 1996.

[PGLK97] A. Pellenkoft, C. Galindo-Legaria, and M. Kersten. The complexity of transformation-based join enumeration. In *Proceedings of the Twenty-Third International Conference on Very Large Data Bases*, pages 306–315, Athens, Greece, August 1997.

[PGMU96] Y. Papakonstantinou, H. Garcia-Molina, and J. Ullman. Medmaker: A mediation system based on declarative specifications. In *Proceedings of the International 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.

[PHH92] H. Pirahesh, J. Hellerstein, and W. Hasan. Extensible/rule based query rewrite optimization in starburst. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 39–48, San Diego, California, June 1992.

[PSC84] G. Piatetsky-Shapiro and C. Connell. Accurate estimation of the number of tuples satisfying a condition. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 256–276, Boston, MA, June 1984.

[QRS+95a] D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, and J. Widom. Querying semistructured heterogeneous information. Technical report, Stanford University Database Group, 1995. Document is available as ftp://db.stanford.edu/pub/papers/querying-full.ps.

[QRS+95b] D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, and J. Widom. Querying semistructured heterogeneous information. In *Proceedings of the Fourth International Conference on Deductive and Object-Oriented Databases*, pages 319–344, Singapore, December 1995.

[RCK+95] Nick Roussopoulos, Chungmin M. Chen, Stephen Kelley, Alex Delis, and Yannis Papakonstantinou. The Maryland ADMS Project: Views R Us. *Bulletin of the TCDE*, 18(2):19–28, June 1995.

[RK95] S. Ramaswamy and P. Kanellakis. OODB indexing by class-division. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 139–150, San Jose, California, May 1995.

[RNS96] Michael Rys, Moira C. Norrie, and Hans-Jörg Schek. Intra-Transaction Parallelism in the Mapping of an Object Model to a Relational Multi-Processor System. In *Proc. VLDB*, pages 460–471, Mumbai (Bombay), India, September 1996.

[RR98] J. Rao and K. Ross. Reusing invariants: A new strategy for correlated queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 37–48, Seattle, Washington, June 1998.

[Run92] Elke A. Rundensteiner. MultiView: A Methodology for Supporting Multiple Views in Object-Oriented Databases. In *Proc. VLDB*, pages 187–198, Vancouver, Canada, August 1992.

[SAC+79] P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price. Access path selection in a relational database management system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 23–34, Boston, MA, June 1979.

[SAD94] Cassio Souza, Serge Abiteboul, and Claude Delobel. Virtual Schemas and Bases. In *Proc. EDBT*, pages 81–94, Cambridge, U.K., 1994.

[SG98] A. Silberschatz and P. Galvin. *Operating System Concepts*. John Wiley and Sons, New York, New York, 1998.

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

[SLT91] Marc H. Scholl, Christian Laasch, and Markus Tresch. Updatable Views in Object-Oriented Databases. In *Proc. DOOD*, pages 189–207, Munich, Germany, December 1991.

[SMY90] W. Sun, W. Meng, and C. T. Yu. Query optimization in object-oriented database systems. In *Proceedings of the International Conference on Database and Expert Systems Applications*, pages 215–222, Vienna, Austria, August 1990.

[SO95] D. D. Straube and M. T. Ozsu. Query optimization and execution plan generation in object-oriented database systems. *IEEE Transactions on Knowledge and Data Engineering*, 7(2):210–227, April 1995.

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

[Suc96] Dan Suciu. Query Decomposition and View Maintenance for Query Languages for Unstructured Data. In *Proc. VLDB*, pages 227–238, Mumbai (Bombay), India, September 1996.

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

[Swa89] A. Swami. Optimization of large join queries: Combining heuristics and combinatorial techniques. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 367–376, Portland, Oregon, May 1989.

[Ull88] J. Ullman. *Principles of Database and Knowledge Base Systems.* Computer Science Press, Rockville, Maryland, 1988.

[Ull89] J.D. Ullman. *Principles of Database and Knowledge-Base Systems, Volumes I and II.* Computer Science Press, Rockville, Maryland, 1989.

[WC96] J. Widom and S. Ceri. *Active Database Systems: Triggers and Rules for Advanced Database Processing.* Morgan Kaufmann, San Francisco, California, 1996.

[Wid95] J. Widom. Research problems in data warehousing. In *Proceedings of the Fourth International Conference on Information and Knowledge Management*, pages 25–30, November 1995.

[Wie92] G. Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, 25(3):38–49, March 1992.

[XH94] Z. Xie and J. Han. Join index hierarchies for supporting efficient navigations in object-oriented databases. In *Proceedings of the Twentieth International Conference on Very Large Data Bases*, pages 522–533, Santiago, Chile, September 1994.

[YM98] C. Yu and W. Meng. *Principles of Database Query Processing for Advanced Applications.* Morgan Kaufmann, San Francisco, California, 1998.

[ZG98] Yue Zhuge and Hector Garcia-Molina. Graph Structured Views and Their Incremental Maintenance. In *Proc. ICDE*, 1998.