Optimizing Information Mediators By Selectively Materializing Data

by

Naveen Ashish

A Dissertation Presented to the

FACULTY OF THE GRADUATE SCHOOL

UNIVERSITY OF SOUTHERN CALIFORNIA

In Partial Fulfillment of the

Requirements for the Degree

DOCTOR OF PHILOSOPHY

(Computer Science)

March 2000

Dedication

This thesis is dedicated to my Mom and Dad - Leela Srivastava and Suresh Srivastava, for stressing right from the beginning the importance I should pay to education, being encouraging and enthusiastic about all endeavours I undertook in this regard, and being "with me" in every sense in all phases of this PhD journey.

Abstract

Building information mediators (a.k.a. information agents) has been a very active area of research in the information and data management community to address the problem of integrating data from multiple sources such as databases or Web sources. An issue with such mediators, particularly Web based mediators is that the speed of any mediator application is heavily dependent on the remote sources being integrated, with often a very large amount of time being spent in retrieving data from the remote sources.

I present an approach for optimizing the performance of information mediators by locally materializing data. I present a framework for materializing data in a mediator environment. The data is materialized selectively. I then present an approach for automatically identifying the portion of data to materialize by considering several factors such as the distribution of user queries, structure of Web sources and updates at the sources. I present experimental results demonstrating the effectiveness of my approach using a materialization system that I implemented for the Ariadne mediator based on the ideas in my thesis. I discuss how my work relates to previous work in materialization and caching in database and Web server environments and also how my approach is applicable to several other mediator systems. Finally I outline some directions for future work in this area.

Acknowledgements

First I would like to acknowledge the support of my thesis advisor Craig Knoblock. He has constantly provided excellent guidance and encouragement on my research work and academic matters. His attitude towards my shortcomings or occasions when I turned up with half baked ideas in the course of this research has been extremely constructive with prompt suggestions about what would be a way to improve or better way to proceed. Finally I thank him for patiently listening and providing advice on problems I occasionally brought before him as a graduate student that were non academic/research matters.

Cyrus Shahabi provided most valuable guidance as a co-advisor. I particularly benefitted from his expertise in core database areas. Besides advice on research issues he also readily provided advice and support for my other endeavours such as applying for and selecting the right job after finishing.

It has been a privelege and a thoroughly enjoyable experience of having done my PhD work at USC's Information Sciences Institute (ISI). I feel few places can offer an environment so conducive to research and such a fun place to work ! I must thank Yigal Arens who gave me this opportunity of coming to ISI as a Research Assistant on the SIMS project and all the ISI staff particularly that of the Intelligent Systems Division for making this a great place to work.

I was fortunate to be a member of the SIMS and Ariadne project groups on information integration. I have learned a lot in all these years. I have on numerous occasions sought help on different aspects of our project from various group members. Each of them - Dan, Greg, Jean and Maria has been always willing and helpful for my thesis work. I must separately mention Andrew Philpot whose help I probably took the most on system aspects, especially considering that he is mostly busy with development responsibilities for our project. Jose Luis Ambite has been an excellent office mate. He was ever willing to be of help on research related questions and we also had many lively discussions on general topics. Besides our project members I must also mention Yolanda Gil who as PhD coordinator took an active interest in and provided advice on general issues to PhD students as we proceeded through the program.

My research work has benefitted tremendously from the advice of other members of my thesis committee. Dennis McLeod has provided me with excellent guidance right from the stage when I was looking for a thesis topic to the present completion. Steve Minton regularly provided useful feedback both as a project member of Ariadne as well as a thesis committee member. Finally I must thank Dan O Leary for his time and valuable suggestions from an MIS perspective.

I am grateful to Richard Hull who gave me the opportunity of visiting Bell Laboratories in the summer of 1997. Under him and Gang Zhou I was able to work in an area of my interest and also got my first experience at a major corporate research laboratory. I also thank other researchers outside USC and ISI who have helped with my research work, that includes Alon Levy, Len Seligman and Maurizio Lenzerini.

My years as an undergraduate in Computer Science at the Indian Institute of Technology Kanpur were extremely fruitful laying the foundation for what I could further do as a graduate student. I still owe my country the debt of that education and pledge to repay it someday.

Finally a note of appreciation to all my friends at USC that were here with me all these years and made life in school and LA fun - that includes Anoop, Arnab, Dibyo, Nishan, Rohit, Satish, Sridevi, Sumathi, Vijay and Vishwesh.

The research reported here was supported in part by the Integrated Media Systems Center, a NSF Engineering Research Center, in part by the Rome Laboratory of the Air Force Systems Command and the Defense Advanced Research Projects Agency (DARPA) under contract number F30602-98-2-0109, in part by the United States Air Force under contract number F49620-98-1-0046, in part by research grants from NCR and General Dynamics Information Systems, in part by cash/equipment gifts from JPL/NASA contract number 961518 and NSF grants EEC-9529152 and MRI-9724567. The views and conclusions contained in this thesis are the author's and should not be interpreted as representing the official opinion or policy of any of the above organizations or any person connected with them.

Contents

List Of Tables

List Of Figures

Chapter 1

Introduction

One of the frontier areas being pursued by database and information management researchers for the past several years is building *information mediators* [Wiederhold, 1992] a.k.a. *information agents* which are systems that can extract and integrate data from multiple databases or semi-structured Web sources. The representative systems include TSIMMIS [Hammer *et al.*, 1995], Information Manifold [Ives *et al.*, 1999], The Internet Softbot [Etzioni and Weld, 1994], InfoSleuth [Bayardo *et al.*, 1996], Infomaster [Genesereth *et al.*, 1997], DISCO [Tomasic *et al.*, 1997], HERMES [Adali *et al.*, 1997], SIMS [Arens *et al.*, 1996] and Ariadne [Knoblock *et al.*, 1998b]. The Ariadne project at USC/Information Sciences Institute is concerned with building the technology and tools for quickly putting together a mediator or information agent application that provides integrated structured query access to multiple prespecified Web sources.

For instance, consider a mediator application on information about countries (hereby referred to as 'the countries mediator') that provides integrated access to Web sources of information about countries in the world. For the countries mediator, the set of sources we provide access to is:

- The CIA World Factbook[1] which provides interesting information about the geography, people, government, economy etc. of each country in the world.
- The NATO homepage[2] from which we can get a list of NATO member countries.
- The InfoNation[3] source which provides statistical data about UN member countries.

We can use such a mediator application to answer interesting queries such as *"Find the defense expenditure and spending on education of all countries that have a national*

[1] http://www.odci.gov/cia/publications/factbook/country.html
[2] http://www.nato.int/family/countries.htm
[3] http://www.un.org/Pubs/CyberSchoolBus/infonation/e_infonation.htm

product greater than 500 billion dollars." Note that it would be far more tedious and time consuming for the user to gather this information himself, given that he can only browse individual sources one at a time. As another example consider TheaterLoc [Barish *et al.*, 1999] which is also an Ariadne mediator application. TheaterLoc provides integrated access to Web sources about movies and theatres, an interactive map server depicting their various locations and a video server from which users can see video trailers of movies playing at the selected theatres. This application is available online at http://www.isi.edu/ariadne

Integrated access is provided to the following Web sources:

- http://www.cuisinenet.com (Cuisinenet) Information about restaurants in various US cities.
- http://movies.yahoo.com/movies (Yahoo Movies) Theater and movie showtime information.
- http://www.hollywood.com (Hollywood.com) Movie previews source.
- http://www.geocode.com (E-TAK Geocoder) Geocodes street addresses.
- http://tiger.census.gov/cgi-bin/mapbrowse-tbl (US Census Map Server) Online interactive map server.

A typical query to such an application might be *"Find all the restaurants and theaters in Beverly Hills and plot them on a map of that area."*

The research problems in building information mediators that have been addressed include information modeling [Knoblock *et al.*, 1998b] for describing contents of different sources being integrated and also the integrated 'view' of information provided over the various sources, query planning to efficiently generate high quality plans for retrieving and integrating data from various sources [Ambite and Knoblock, 1998] and semi-automatically generating *wrappers* that allow database like querying of semi-structured Web sources [Ashish and Knoblock, 1997, Muslea *et al.*, 1998].

An issue with building applications using Ariadne or other similar mediators is that the speed of the resulting application is heavily dependent on the data sources. The response time may be high even if the mediator query planner generates high quality information gathering plans. This is mainly because to answer many queries a large number of Web pages may need to be fetched over the network or the Web sources may be slow. Without any kind of optimization i.e., assuming that all data must be fetched from the Web sources in real time, the above query to the countries mediator *"Find the defense expenditure and*

spending on education of all countries that have a national product greater than 500 billion dollars" can take several minutes to return an answer. This is because for this particular query the mediator must retrieve the pages of all countries in the CIA World Factbook to determine which ones have a national product greater than $500 billion, which takes a large amount of time. Without optimization, a typical query to TheaterLoc such as *"Find all the restaurants and theaters in Beverly Hills and plot them on a map of that area."* also takes several minutes to return an answer. For this query, the mediator first finds the restaurants and theaters in Beverly hills (from the Web sources such as `Zagats` and `Yahoo movies`) and then geocode each restaurant and theater so as to be able to plot it on a map of the area. Geocoding the theaters and restaurants takes a large amount of time as the Geocoder source is structured such that we can only geocode one restaurant or theater at a time. The above examples are typical of most mediator applications where to answer a query, often the time spent in gathering the data from the remote Web sources is very high. This is for several reasons, either a large number of Web pages have to be retrieved to answer the user query, the Web sources are structured such that retrieving data is time consuming (e.g., the Geocoder), or a particular Web server is slow i.e., retrieving even a single page can be very time consuming (e.g., the map server can take as much as two minutes to return a single map).

An obvious approach to improving performance is to locally materialize data at the mediator instead of retrieving it from the remote sources each time. This dissertation presents a systematic approach to optimizing the performance of information mediators by locally materializing data. The first contribution is presenting a framework for materializing data in a mediator environment. I then provide an argument for materializing data *selectively*. Finally I present our approach to automatically identifying the portion of data that must be materialized considering a combination of several factors. I now present the outline of my approach in more detail.

1.1 Approach

The brute force approach would be for each mediator application to simply materialize locally all the data in all the Web sources being integrated. This would no doubt speed up the application considerably. However materializing all the data is impractical for several reasons. First, the amount of space needed to store all the data locally could be very large. Also we must take into account the fact that the data can get updated at the original Web sources and the cost of keeping the materialized data consistent could be

very high. In fact by locally storing all the data, the mediator really degenerates into a data warehouse, defeating the purpose of a mediator that provides *access* to information in different sources (and can scale to a large number of sources) rather than storing it all locally. Finally, it is our hypotheses that a significant performance gain can be achieved by just materializing a small fraction of the data that the mediator can access. We make this claim for two main reasons. First, for many mediator applications it is likely that users will be more interested in querying some portions of the data than others. Thus we could just materialize the portion of data frequently queried. Next, Web sources are structured such that certain types of queries can be very expensive. In many cases we could materialize just a small portion of data (explained more in detail later) that can significantly speed up the processing of the expensive queries. We thus argue that data must be *selectively* materialized.

In the approach to optimizing mediator performance by selectively materializing data there are two primary issues that must be addressed. First, what is the overall framework for materializing data i.e., how do we represent and use the materialized data ? Next, having argued that data must be selectively materialized, how do we automatically identify the portion of data that is most useful to materialize? We now discuss our approach to these problems.

1.1.1 Materialization Framework

For any kind of performance improvement system based on locally materializing or caching data we need a framework for representing and using the materialized data. For instance for database system caches we have page based schemes, tuple based schemes or recent approaches based on *semantic* caching [Dar *et al.*, 1996, Keller and Basu, 1996]. For the mediator environment I have built upon an idea originally described in [Arens and Knoblock, 1994]. The basic idea is to locally materialize useful data and define it as another information source for the mediator. The representation and use of materialized data is done in a manner similar to that in semantic caching in databases where a semantic description of the cached or materialized data is provided. The system reasons with the semantic description to determine what portion of a query can be answered using the materialized data.

I will first provide an overview of the Ariadne architecture with particular emphasis on information modeling in information mediators, followed by a description of the overall approach. The Ariadne architecture [Knoblock *et al.*, 1998a] borrows heavily from that of SIMS, SIMS is used to integrate data from mainly database systems whereas Ariadne

is focussed on integrating semi-structured Web sources. In fact the SIMS architecture is typical of several other mediator systems such as Information Manifold [Ives *et al.*, 1999], InfoSleuth [Bayardo *et al.*, 1996], Infomaster [Genesereth *et al.*, 1997] etc.

In the SIMS system we use the LOOM knowledge representation language [MacGregor, 1988] (we can also view this as a data model) for modeling data. The user is presented with an integrated view of the information in several different sources which is known as the *domain model*. We describe the contents of the individual information sources in terms of the domain model. A simple example is shown in Figure 1.1 (a). The white circle labeled COUNTRY represents a domain *concept* (equivalent of a class in an object-oriented model) and the shaded circles represent sources. The arrows on the circles represent attributes of the concepts. In this example the domain concept COUNTRY provides an integrated view over two sources of information about countries – FACTBOOK-COUNTRY and INFONATION-COUNTRY. The user queries the integrated view i.e., concepts in the domain model and the query planner in the mediator generates plans to retrieve the requested information from one or more sources. Please refer to [Arens *et al.*, 1996] for a more detailed description of SIMS.

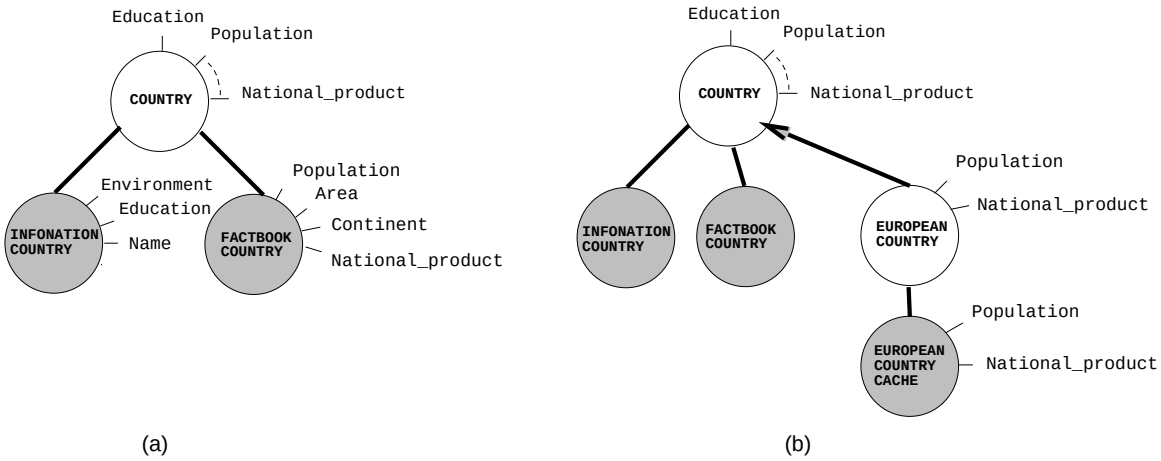


Figure 1.1: Information modeling in SIMS

We identify useful classes of information to materialize, materialize the data in these classes in a database local to the mediator, and define these classes as auxiliary information sources that the mediator can access [Ashish *et al.*, 1998, Ashish *et al.*, 1999]. For instance in an information about countries application[4] suppose we determined that the class of

[4]The model showing the attributes of the COUNTRY concept in the countries application is given in the appendix. We will be using this model in examples throughout the paper

information - *the population and national product of all European countries* was frequently queried and thus useful to materialize. We materialize this data and define it as an additional information source as shown in Figure 1.1 (b). Given a query the mediator prefers to use the materialized data instead of the original Web source(s) to answer the query.

1.1.2 Selecting Data to Materialize

We are now left with the question of identifying what is the portion of data that is most useful to materialize. There are several factors that can be analyzed or must be considered for identifying such data. The classes of data that are most frequently queried by users are obviously good candidates for materializing locally. One of the factors we can analyze thus is the distribution of previous user queries to the mediator to determine the frequently accessed classes of data. Another factor that must be taken into account is the structure of Web sources being integrated. We provide database like query access to Web sources that were not originally designed for structured querying by building wrappers around the sources. As a result certain kinds of queries can be very expensive to execute. For instance consider a query to the `CuisineNet` Web source where we ask for *"the names and telephone numbers of all chinese restaurants with a service rating of 9 or above."* The source is structured such that the only way to answer the query is to retrieve pages of *all* chinese restaurants in `CuisineNet` to determine which ones have a service rating of 9 or above and this obviously takes a long time. In many cases we can prefetch and materialize data that will help improve the response time of expensive queries. For instance in the above example we could locally materialize the names and service ratings of all restaurants in `CuisineNet` and thus determine the ones satisfying a certain rating by looking at just the local data i.e., without having to scan pages of all chinese restaurants. Finally we have to take into account the fact that the sources can get updated. As the user must be provided with consistent data, we must also incorporate the cost of keeping the materialized data up to date when selecting classes to materialize.

To summarize the above, the following are the factors we consider for selecting data to materialize:

- The distribution of user queries
- Structure of sources
- Updates at Web sources

We now elaborate a little on how we analyze these factors.

1.1.2.1 Analyzing the Distribution of User Queries

We analyze the distribution of user queries to determine the most frequently accessed classes of data. We look for "patterns" in query distributions. A pattern may be a set of frequently queried attributes in a class. For instance in the COUNTRY class it may be that the set of attributes area, population, national product is very frequently queried. A pattern may also be a set or subset of objects (tuples) in a class that are frequently queried. For instance in the COUNTRY class it may be that *Asian* countries are queried most frequently. The above correspond to vertical and horizontal partitions of the data respectively. Finally we may have *associations* between classes and attributes. For instance a patterns such as the area, population of European countries that is queried very frequently.

We have developed an algorithm for extracting such patterns from a user query distribution. A key feature of this algorithm is that it outputs a *compact* description of the patterns extracted. A compact description is necessary from a performance perspective and we shall elaborate on this in Chapter 3. The algorithm first classifies queries by analyzing constraints in the queries to determine what classes of data the user is interested in. We construct an ontology of such classes that users are interested in. For instance users may be interested in *European Countries* or *Democratic Countries*, etc. For each such class we then determine what groups of attributes are frequently queried. For instance for European Countries, users may be primarily interested in say the *economy* and *national product*. We try to merge together attribute groups queried with approximately the same frequency to make the description more compact. Finally we can further make the description more compact by merging classes based on class covering relationships. The algorithm thus tries to compactly describe classes of data that are present as patterns in a query distribution. For instance the algorithm may extract patterns such as users are interested in *"the national product and economy of all European countries"*.

1.1.2.2 Analyzing the Structure of Sources

In Web-based mediators we provide database like query access to semistructured Web sources by building wrappers around the sources. Often the Web sources have limited querying capabilities. As a result, certain kinds of queries can be very expensive as the query functionality not provided by the source is provided by the wrapper or the mediator.

For each Web source being integrated we can determine in advance what are the expensive classes of queries that could be asked in the application. We can then materialize data that could improve the response time for the expensive kinds of queries.

In our approach, we start with a specification of the query interface to a mediator application that defines exactly the kinds of queries a user could ever ask in that mediator application. We then estimate the costs of the various classes of queries using a cost estimator which is part of the mediator. The purpose is to identify in advance the expensive kinds of queries that could be asked in a particular mediator application. Then, using heuristics based on the kind of query, source or sources used to answer the query and also the different data processing operations that are performed to answer the query, we prefetch and materialize data that can improve the response time for the expensive query.

For instance consider a mediator application that includes amongst other sources an online *geocoder* that accepts street addresses of places and returns the latitude and longitude of the place. Now geocoding (converting street addresses to latitudes and longitudes) a set of addresses using this source is an expensive query since the source is structured such that we can only geocode one address at a time. Further the mediator application might be such that we only geocode a fixed set of places every time (for instance the set of restaurants in LA). In such a case we should materialize the result of geocoding this set of places even before analyzing any set of user queries to the mediator.

1.1.2.3 Updates and Maintenance Cost

Finally we must address the issue of updates at Web sources. First the materialized data must be kept consistent with that in the Web sources. It may be that the user is willing to accept data that is not the most recent in exchange for a fast response to his query (using data that is materialized). Thus we need to determine the frequency with which each class of data materialized needs to be refreshed from the original sources. Next the total maintenance cost (the maintenance cost for a class of data is the time spent by the mediator in refreshing the materialized class each day) for all the classes of data materialized must be kept within a limit that can be handled by the system.

We have developed a language for describing the update characteristics and frequency of updates for various source classes and attributes and also the user's requirements for freshness of each domain class and attribute. We illustrate the kinds of characteristics we can specify using an example. Consider an information source for movies which we model as a source relation MOVIE_SRC having attributes such as theatre, showtimes, actors,

director, review, etc. Table 1.1 shows the update specification for the movies relation and some attributes, the specification is implemented as a database relation.

CLASS	MEMBERSHIP	CHANGE	TIME_PERIOD	TIME
MOVIE_SRC	A	Y	1 week	week:friday

Table 1.1: Characteristics of updates for a source class

It states that for the MOVIE_SRC source class, the MEMBERSHIP= 'A' i.e., arbitrary (instances of the class can be added or deleted), CHANGE = 'Y' i.e., yes (values of objects or class members can change), TIME_PERIOD is 1 week so the data changes once every week and the TIME of change is every friday. The attributes CHANGE, TIME_PERIOD and TIME also characterize each attribute of a source class. By default for each attribute in a source class the values of the update characterization attributes are propagated from the source relation they are part of. So for instance the attribute 'theatre' has the characteristics CHANGE= 'Y', TIME_PERIOD= 1 week, etc. These defaults can be overridden by explicitly stating the new values. So for instance we could specify the characteristics for the 'actors' attribute as shown in Table 1.2 since the value of actors for MOVIE_SRC does not change.

ATTRIBUTE	CHANGE	TIME_PERIOD	TIME
actors	N	-	-

Table 1.2: Characteristics of updates for an attribute

For each domain class we also specify for each attribute the user's requirements for currency of data, actually stated in terms of how stale he can tolerate the data to be. Consider a domain class called MOVIE where we integrate information from several sources about movies.

ATTRIBUTE	TOLERANCE
theatre	0
showtimes	0
actors	0
director	0
review	6 weeks

Table 1.3: Staleness tolerances for attributes of a domain class

Table 1.3 shows the user's tolerance for various attributes of the domain class MOVIE. It states that the value of attributes such as 'theatre' and 'showtimes' must be current (having a tolerance of 0) whereas the 'review' can be up to 6 weeks old.

We have developed an algorithm that using the above specifications can determine the frequency with which attributes in each materialized class must be updated to be

consistent with the user's requirements. It also computes the maintenance cost for that class. The maintenance cost is considered in two ways. First, for each individual class if the maintenance cost is very high we may decide not to materialize it. For instance a class having a *stock quote* as one of its attributes which is updated every 5 seconds has a very high maintenance cost and it may be better not to materialize such a class at all. Second, the total maintenance cost for all the classes of materialized data should be kept within a limit that can be handled by the system.

1.1.2.4 Admission and Replacement

Any materialized class occupies some local space and has an associated maintenance cost. For any mediator application, we assume that we will have a limited total space for storing the materialized classes of data and also a limited total maintenance cost that can be borne for keeping the materialized classes up to date. It may be that we have several classes of data that we want to materialize (based on query distribution analysis and source structure analysis) but we do not have the space or cannot bear the maintenance cost for all the classes. We have developed a method for quantitatively ranking proposed materialized classes. We assign a *profit* to each class by considering factors such as expected query response "savings" due to each materialized class, space occupied by each class, maintenance cost, etc. This is discussed in detail in Chapter 6.

Also for each application, the set of classes that are optimal to materialize can change over time. In particular, the classes of data queried most frequently can change over time. For instance in TheaterLoc the set of movies users query most frequently or even the set of restaurants queried most frequently will change over time. As patterns change we periodically re-evaluate the query distribution to determine the frequently accessed classes in the most recent query distribution. We have also developed an admission and replacement policy where new classes of data may be materialized locally and existing classes thrown out depending on the relative benefits of the classes.

1.1.2.5 An Integrated Materialization System

Based on the above ideas we have developed an "integrated materialization system" for optimizing mediator performance. We define this integrated materialization system to be the complete software system which performs all the tasks for optimizing mediator performance by data materialization. It includes subsystems for selecting data to materialize based on query distribution analysis, source structure analysis and update aspects, a local

database system for storing the materialized data, and a subsystem for keeping the materialized data consistent. The materialization system should be viewed as a stand alone "plug-in" to a mediator that can interact with the mediator and optimize its performance.

There are several issues that we had to address in putting together the above ideas to form a complete materialization system. One of the issues is in what order (if any) should the various factors be analyzed ? The maintenance cost must always be taken into account for any class of data materialized. In fact the maintenance the cost for a particular data item may be very high and thus we may decide not to materialize any class containing that data item. Thus the update costs should be analyzed before the source structure or query distribution. Based on source structure analysis and with estimates of update costs we may decide to materialize a particular class even before looking at the user query distribution. Thus source structure should be analyzed before the query distribution. We have developed a systematic framework that defines exactly how various factors are analyzed, how they affect one another and how various factors are combined to determine what classes of data are to be materialized. Another issue is that the materialization system must periodically refresh the materialized classes at appropriate time intervals. The refreshing frequency is determined by analyzing the update characteristics and user requirements. Finally the materialization system needs to make changes to the domain and source models of a mediator application to register the changes due to the new materialized classes defined as auxiliary information sources.

1.1.3 Contributions

The major contributions of this thesis can be summarized as follows:

- I have presented a framework for materializing data in mediators by defining the materialized data as another information source.
- Arguing that data must be materialized selectively, I have presented an approach for automatically identifying data to materialize based on considering several factors.
- I developed an algorithm for extracting patterns from a distribution of user queries which compactly describes frequently queried classes of data.
- I developed an approach for systematically analyzing the structure of Web sources to predetermine expensive queries with heuristics to prefetch data that can improve response time for the expensive queries.

- I developed an approach for specifying update characteristics of Web sources with user requirements for freshness. I estimate the maintenance cost for each materialized class of data from the update specifications and incorporate it into the decision of selecting data to materialize.

- Finally, I put together the above ideas to develop a complete materialization system that can augment a mediator to optimize its performance.

1.2 Organization

The remainder of this thesis is organized as follows. In Chapter 2, I provide a high level overview of the design of the materialization system. In Chapter 3, I describe how we extract patterns from a distribution of user queries. In Chapter 4, I describe my approach for analyzing the structure of sources for prefetching data. In Chapter 5 I describe update aspects. I present a specification language for update characteristics and user requirements for freshness of data. I also describe an algorithm for estimating maintenance cost. In Chapter 6, I describe how all the above factors are pieced together to form the complete materialization system. I present integration issues, the admission and replacement policy for materialized data classes and the details of the interaction of the materialization system with the mediator. In Chapter 7, I present experimental results for the materialization system that I implemented for the Ariadne information mediator. I tested the system in 3 different mediator applications that we have developed for Ariadne recently. In Chapter 8 I discuss related work especially in the materialization and caching areas in the database, operating systems and Web server domains. I also provide a detailed discussion on the generality of the materialization approach with regards to applying it to several other mediator systems. Finally in Chapter 9, I discuss directions for future work and provide a conclusion.

Chapter 2

Materialization System Overview

We provide a high level overview of the materialization system. We will be presenting a modular architecture for the system with different tasks such as query distribution analysis, source structure analysis, update analysis etc. allocated to individual modules.

2.1 A Modular Architecture of the Materialization System

To review the definition of the materialization system provided in the introduction, it is the complete software system doing the tasks of query distribution analysis, source structure analysis, update aspects, combining all the above factors for selecting data to materialize and including a local database system for finally storing the materialized data.

We have designed the materialization system as having a modular architecture with separate modules for performing the various tasks of query distribution analysis, source structure analysis, etc. as illustrated in Figure 2.1. The primary reasons for this architecture have mostly to do with the advantages of a modular architecture over a monolithic architecture for any large software system in general. The various tasks of the materialization system are distinct and clearly separable. A modular architecture provides a means for assigning separate tasks to separate modules. This provides a more understandable conceptual view of the system and makes maintenance or enhancements to various parts easier.

We have 5 different modules in the system, namely:
(i) Query Distribution Analysis (extracting patterns from queries)
(ii) Source Structure Analysis
(iii) Update Analysis
(iv) Admission and Replacement
(v) Materialization

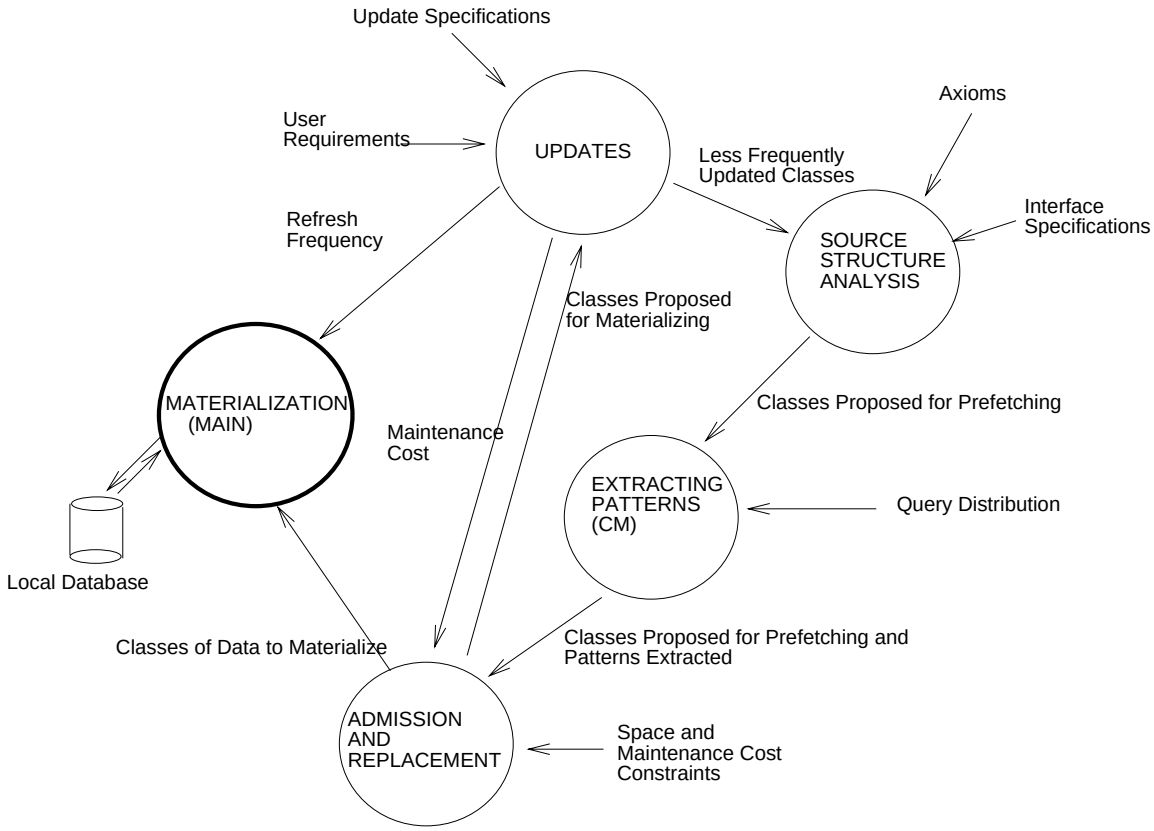


Figure 2.1: Various Modules of the Materialization System

We now describe briefly what task each module performs. We also describe the input to and output from each module and also the data flow between these modules. We will proceed in the order in which the modules perform their tasks once the materialization system starts to optimize a new mediator application. (i) Update Module: The primary task of this module is to compute the frequency of update and maintenance cost for any class of materialized data given the update specifications. For optimizing any new application, first the update module uses the update specifications to determine what classes of data need to updated very frequently if materialized. This is so that we can remove such classes from consideration for materialization. For classes of data that we do consider materializing, the maintenance cost for each materialized class must also be estimated to compute the profit for that class. At a later stage thus the update module also provides estimates of the maintenance cost for any class. To summarize:
Inputs: Update specifications, Classes proposed for materialization
Outputs: Classes that must not be considered for materialization because of very high update frequency, Maintenance cost estimate for each proposed materialized class.

(ii) Source Structure Analysis: We then proceed to the source structure analysis module.Its task is to identify expensive kinds of queries and then using knowledge of the structure of Web sources to identify data that if prefetched and materialized will speed up the response time of these queries. It uses the mediator GUI specification to determine what queries can be asked in a particular mediator application. It uses estimates from the query cost estimator in the mediator to determine which of these queries can be expensive. Finally it uses query processing *axioms* to identify data that if materialized will improve response time for these expensive queries. The axioms are precompiled axioms that tell how data for a particular domain class is obtained by a source or combination of sources. The axioms essentially encode what data processing operations (select, join etc.) will be performed across sources and also what query processing limitations are present in sources (e.g., binding patterns). To summarize:
Inputs: Set of classes, GUI specification, Query cost estimates, Query processing axioms.
Output: Set of classes proposed for materialization.

(iii) Query Distribution Analysis Module: We then move on to the query distribution analysis module. The task this module performs is to extract patterns of frequently accessed classes from a user query distribution. To summarize:
Input: Set of classes, User query distribution
Output: Set of classes (proposed for materialization) that are frequently accessed classes in the query distribution.

(iv) Admission and Replacement Module: As mentioned in the Introduction chapter, our approach is to materialize data selectively. Both the source structure and query distribution analysis modules propose classes of data to materialize. However it may not be possible to materialize all these classes because of limitations of space or resources for keeping the materialized data up to date at the mediator side. The task of the admission and replacement module is to optimally select classes of data to materialize (from classes of data proposed for materialization by source structure analysis and query distribution analysis) given constraints of space and maintenance cost at the mediator side. It also makes decisions regarding replacement of materialized classes. It is likely that for any mediator application the patterns in the query distribution may change with time. In this case newer classes of data are queried more frequently and the materialization system must materialize these new classes and throw out existing materialized classes that are not queried frequently any more. To summarize:
Inputs: Classes of data proposed for materialization, Estimates of maintenance cost, Space occupied by each class, System constraints of space and maintenance cost.
Output: Classes of data that will be materialized.

(v) Materialization Module: This should be viewed as the 'main' materialization module. It uses the services of the other modules and interfaces to the local database for performing the overall materialization task. The admission and replacement module hands it a set of classes that must be materialized. It then retrieves the data for these classes and materializes it in the local database. It also interfaces to the mediator application so that the mediator application can use the materialized data. Finally it also refreshes the materialized classes of data at appropriate intervals.

The above has just been a brief summary of the tasks of each of the modules. In the subsequent chapters we describe in detail how exactly each module performs its assigned task. We now describe some more high level issues regarding the interaction between the various modules.

2.2 Interaction Between Modules

Although the tasks of various modules are clearly defined and separable, there is of course interaction between the modules to perform the overall materialization task. We discuss the important issues regarding interaction between different modules, namely order of analysis by the modules and the details of the data flow between modules.

2.2.1 Order of Analysis by Various Modules

One of the issues is to determine in what order (if any) must the various factors be analyzed. There may be some class(es) or attribute(s) that we decide not to materialize just based on the fact that the update frequency is extremely high. For instance we decide not to materialize a stock quote that is updated every 5 seconds. Assuming that the user's tolerance for staleness is 0 (as it quite likely will be for such data), any stock quote(s) materialized will have to be updated every 5 seconds which will be quite expensive. Obviously then we would want to determine the update frequencies for various attributes before either prefetching them based on source structure analysis or materializing them if they happen to be extracted amongst frequently queried classes. We must also do structure analysis before analyzing the query distribution. This is because we may choose to consider some classes for materializing just by analyzing the structure of sources. In that case we do not need to look for patterns for those classes in the distribution of queries.

2.2.2 Data Flow Between Modules

We now discuss more specifically the data flow between the various modules in the performance optimization system. We record information about each attribute in a domain class as we go through the various modules. At any step each domain attribute is annotated as one of 'N','M','D'. 'N' means that we have decided not to materialize that attribute. 'M' means that we have decided to materialize the attribute. 'D' means that we have not yet decided whether or not to materialize it, we are waiting to analyze other factors before reaching a decision. Initially all domain class attributes are annotated as 'D'. We initially analyze the update specifications using the update module to determine what attributes would be updated very frequently. If the update frequency for any attribute is above a certain preset threshold value, we annotate the attribute as 'N'. We then move on to the source structure analysis module. We use the precompiled axioms and the cost estimator to propose classes for prefetching and materializing. Of course none of the attributes in these classes must have been annotated as 'N' previously. The attributes of the classes we propose to prefetch are annotated as 'M'. We then use the query distribution module to extract patterns from the remaining classes i.e., those with attributes still annotated as 'D'. Finally we move on to the admission and replacement module. It takes as input proposed classes to materialize output by both the source structure and query distribution

analysis modules, estimates of maintenance cost from the update module and also specifications of local space and maintenance cost constraints. It then selects classes of data that must finally be materialized by analyzing the costs and benefits of each class.

2.3 Summary

After introducing the issue of speed in mediator applications and outlining our approach to addressing this issue based on local materialization in the previous chapter, we have now given a high level overview of the materialization system. We presented a modular architecture for the overall materialization system. We described briefly the tasks of various modules and also the data flow and interaction between modules. In the following chapters we will describe in detail how each of the modules performs its assigned task.

Chapter 3

Extracting Patterns from Queries

One of the hypotheses of my approach is that there will be *patterns* present in user queries i.e., some classes of data will be queried more frequently than others. Such frequently queried classes of data are of course useful to materialize locally. In this chapter we review the notion of patterns described in the introduction chapter. We then describe how we extract such patterns from user queries. We present an algorithm for extracting patterns from a distribution of user queries along with experimental results demonstrating its effectiveness. We also present a complexity analysis and description of language of patterns learned by the algorithm.

3.1 Patterns and Materialization

As described earlier, a pattern is essentially a class of data that is frequently queried. For instance in TheaterLoc an example of a pattern is *"the address and telephone number of all Mexican restaurants."* This leads us to the problem of how we extract such patterns from user queries. We now present an algorithm that we have developed, called the CM (cluster and merge) algorithm for extracting such patterns from a distribution of user queries.

A key feature of this algorithm is that it outputs a *compact* description of the patterns extracted. A compact description of frequently accessed classes is necessary from a performance point of view. For each class of data we materialize we define a new information source for the mediator. The general problem of query planning to gather information from many sources is combinatorially hard and having a large number of sources will create performance problems for the query planner. With a compact description we keep the number of patterns extracted and thus the number of new information sources created small.

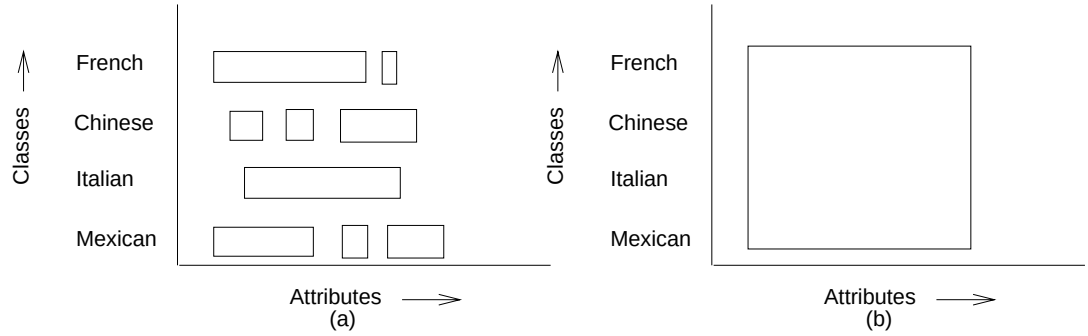


Figure 3.1: Compact Description of Patterns

For instance consider the patterns illustrated graphically in Figure 3.1. The X-axis represents various attributes and the Y-axis represent subclasses. Assume the four subclasses shown on the Y-axis in Figure 3.1 (a) (i.e., French, Italian, Chinese and Mexican restaurants) form a covering of the class 'Restaurant'. The rectangular boxes represent patterns i.e., the class or subclass and the attributes in the pattern. In 3.1 (a) we see 9 different patterns. From a materialization perspective, materializing classes corresponding to these patterns implies creating 9 new information sources in the mediator application. However the single pattern shown in 3.1 (b) represents approximately the same data. In fact it contains all the 9 patterns in 3.1 (a) and some extra data. Materializing the class corresponding to this pattern implies creating just a single new information source.

The CM algorithm is designed such that it extracts compact patterns as illustrated above. It analyzes queries in a distribution one at a time. It determines what classes or subclasses of data users are most interested in by analyzing constraints in the queries. It also determines what attributes or groups of attributes users query most for each class or subclass. Finally it also attempts to *merge* together several classes of data to make the description more compact. We now describe this algorithm in complete detail.

3.2 The CM Algorithm for Extracting Patterns

The CM algorithm takes as input a distribution of user queries and outputs a *compact* description of patterns that extracts from the query distribution.

3.2.1 CM Algorithm

The pseudo code for the CM() algorithm is given in Figure 3.2. There are three main steps in the algorithm:

- Classifying queries (CLASSIFY_QUERIES()). This is to determine what classes of data the user is interested in.
- Clustering attribute groups (CLUSTER_ATTRIBUTE_GROUPS()). To determine attribute groups of interest for each class.
- Merging classes (MERGE_CLASSES()). This is to try and merge classes of data to make the description more compact.

We now describe the steps in the algorithm in more detail.

3.2.1.1 Classifying Queries

We first analyze queries to determine what classes of information users are interested in (procedure CLASSIFY_QUERIES()). For instance queries of the form:

SELECT A
FROM COUNTRY
WHERE region= "Europe"

indicate that the user is interested in the class of European countries. We maintain an ontology in LOOM of classes of information that are queried by users. Initially the ontology contains only the classes in the domain model. We then add sub-classes of these existing classes to the ontology, the sub-classes are generated by analyzing constraints in the user queries. Assuming an SQL syntax for the queries, a query to a domain class has the following general form:

SELECT A
FROM S
WHERE P

where A is the set of attributes queried for the domain class S and $P = P_1$ and P_2 and ... P_n are predicates specifying the query constraints (we restrict ourselves to conjunctive queries). We denote as S_P the "query sub-class" which is the sub-class of S satisfying P. We denote as $\{S_P1, S_P2, ..., S_Pn\ \}$ the set of "sub-classes of interest" where the P_i s are the individual predicates comprising P and S_Pi is the sub-class of S satisfying P_i. For

```
CM (QS) { /* QS is set of user queries */
      O=CLASSIFY_QUERIES(QS) /* build ontology O of classes
of interest */
      CLUSTER_ATTRIBUTE_GROUPS(O) /* cluster attribute
groups in each class */
      FOR (all coverings (S,C) in O) { /* S is the superclass and C set of subclasses in covering */
         MERGE_CLASSES(S,C) /* merge classes */
      }
}

CLASSIFY_QUERIES(QS)
WHILE ( more queries(QS)) {
   Q = get_next_query(QS)
   S_P = get_query_subclass(Q) /* returns subclass of S satisfying P (predicate set) */
   { S_P i } = get_interest_subclasses(Q) /* returns subclasses of S based on individual predicates Pi */
   update_ontology(S_P) /* add any new subclasses appropriately */
   IF P = ∅ { /* P is the predicate set in the WHERE clause */
    update_attribute_count(S,A)
   } ELSE {
    n=count({ S_P i })
    FOR i = 1 to n DO update_attribute_count( S_P i, A ) /* A is the attribute set in SELECT clause */
   }

CLUSTER_ATTRIBUTE_GROUPS(O) {
/* cluster attribute groups for each class in O with similar frequency
and similar attribute groups */
}

MERGE_CLASSES(S,C)
i=1
n=number_of_elements(C) /* size of set C */
totalsize = 0 /* space occupied by matching groups */
seed = get_seed(Ci) /* pick an attribute group from Ci */
WHILE (seed ≠ ∅ ) {
   seedsize=get_size(seed) /* space occupied by seed */
   totalseedsize= seedsize*n /* space occupied by (S,seed) */
   FOR i = 1 to n DO {
     temp = find_match(seed, Ci) /* find best matching group for
seed in Ci */
     size = get_size(temp)
     totalsize += size /* total size of matching groups */
   }
   IF (totalsize/totalseedsize) ≥ MERGETHRESHOLD {
/* merging criteria satisfied */
    remove(seed,C) /* remove attributes in seed from all classes */
    add_group(S,seed) /* form group in superclass */
   }
   ELSE {
     mark_down(seed,Ci) /* so that the same seed is not chosen again */
   }
i= (i+1)mod(n) /* chose next class in C cyclically */
seed = get_seed(Ci) /* get seed from next class */
}
```


Figure 3.2: The CM algorithm for extracting patterns in queries

instance consider a query such as:

SELECT *population, area*
FROM COUNTRY
WHERE (region = "Europe") AND (government = "Republic");

In the above query the query sub-class is that of European Republic Countries and the sub-classes of interest are European Country and Republic Country. In the CLASSIFY_QUERIES( ) procedure for each query we first determine the query sub-classes and set of sub-classes of interest and insert them into the ontology if they are not already present. For instance for a set of queries on the concept COUNTRY in which the WHERE clauses have constraints on the attributes *region* (bound to a value such as Europe, Asia etc.) or *government* (bound to a value such as Republic, Monarchy, Communist etc.) or both, we would create an ontology such as shown in Figure 3.3.(The arcs in the figure represent coverings of groups of subclasses for the superclass COUNTRY).

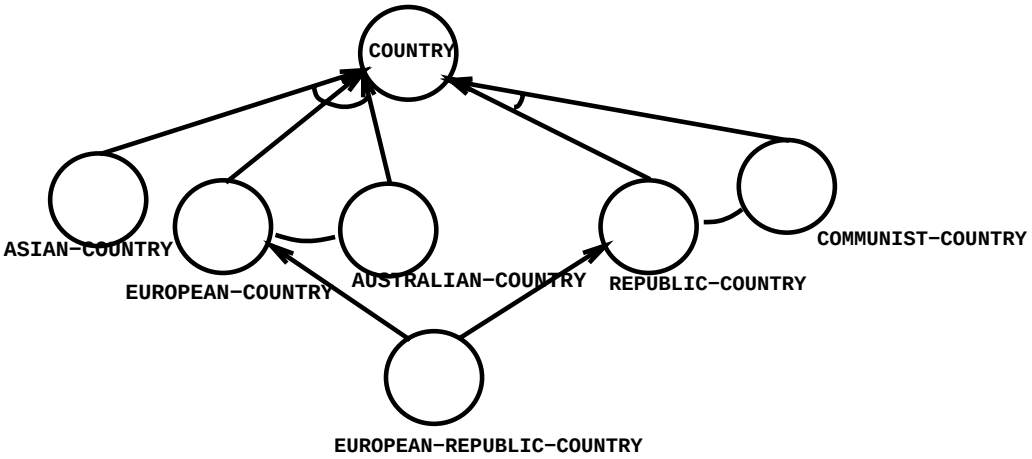


Figure 3.3: Ontology of subclasses of COUNTRY

We also update the query count for the query sub-class S_P for the attribute group A. This is to maintain a record for each sub-class of what attribute groups have been queried and how many times.

3.2.1.2 Clustering Attribute Groups

After the step of classifying queries we have an ontology of classes of interest and also for each class what attribute groups have been queried and with what frequency. We attempt to merge together attribute groups with similar frequencies and with similar attribute groups in order to reduce the number of groups for each class that we have to consider. Attribute groups are considered 'similar' if they have mostly overlapping sets of attributes. The similarity measure is based on the fraction of attributes that are common to both attribute groups. Merging attribute groups makes the description of classes more compact. The problem of merging together attribute groups based on similarities of two factors, namely the similarity of the attribute groups and the similarity of query frequency can be formulated as a two dimensional clustering problem. It is known that the problem of finding an optimal clustering is NP complete [S.J.Wan *et al.*, June 1988, Hyafil and R.L.Rivest, May 1976]. We have developed an approximation algorithm for this purpose. Attribute groups are merged together if the relative difference of their frequencies is within a preset limit known as CLUSTER-FREQUENCY-DIFFERENCE and the similarity of attribute groups is at least GROUP-SIMILARITY-THRESHOLD. This is done in the procedure CLUSTER_ATTRIBUTE_GROUPS( ).

We provide an illustration below. Table 3.1 shows the attribute groups for some class along with the query frequencies for the attribute groups. Table 3.2 shows the results after merging the attribute groups based on similarity of attribute group and query frequency. This results in a more compact description with fewer classes.

attribute groups	frequency
{area,gdp,economy }	9
{coastline,ports }	2
{area,economy,percapita }	11
{gdp,economy,forests }	4
{coastline,ports,airports }	3
{coastline,ports,railways }	2

Table 3.1: Attribute groups

attribute groups	frequency
{area,gdp,economy,percapita }	10
{coastline,ports,airports,railways }	2.3
{gdp,economy,forests }	4

Table 3.2: Merged attribute groups

3.2.1.3 Merging Classes

We mentioned earlier that it is important to keep the number of classes of information materialized small from a query processing perspective. Consider the following classes of information, each of which is essentially a group of attributes in a class:
(i) (EUROPEAN-COUNTRY,{population,area})
(ii) (ASIAN-COUNTRY, {population,area})
(iii) (AFRICAN-COUNTRY,{population,area})
(iv) (N.AMERICAN-COUNTRY,{population,area})
(v) (S.AMERICAN-COUNTRY, {population,area}) and
(vi) (AUSTRALIAN-COUNTRY,{population,area}).
We could replace the above six classes by just one class (COUNTRY,{population,area}) which represents exactly the same data. In general thus a group of classes of information of the form $(C_1, A), (C_2, A), ..., (C_n, A)$ [1] may be replaced by one class i.e., (S, A) if $C_1, C_2, .., C_n$are direct subclasses of S and form a covering of S. As the ontology of classes is maintained in LOOM, we use LOOM to determine groups of classes we can merge based on class/subclass relationships.

In fact we also allow for a kind of 'relaxed' merge where we may merge a set of classes such as $(C_1, A_1), ..., (C_n, A_n)$ to (S, A) where the $C_i s$ are direct subclasses of S as above. However $A_1, ..., A_n$ need not be exactly equal groups rather they just need to overlap, and A is the union of $A_1, ..., A_n$. The disadvantage in this case is that the merged class of information will contain some extra data i.e., data not present in any of the classes of information merged to form the merged class. There is a tradeoff between space wasted to store the extra data and the time gained (in query planning) in reducing the number of classes materialized. The amount of space that can be wasted by extra data is limited by a parameter known as the MERGE-THRESHOLD.

The procedure MERGE_CLASSES( ) shows how we do exact or relaxed merging of classes. The procedure takes as input a superclass S and a set of subclasses C of S that form a covering of S. For each class in C we also have the set of groups of attributes queried. The basic idea is to take an attribute group A in a class C_i in C and see if we can merge with other groups in other classes in C to the group A in the superclass S. We describe the steps in the procedure MERGE_CLASSES( ) by stepping through the procedure with an example as shown in Tables 3.3 and 3.4 . The first column shows the the various classes in C along with their attribute groups. The asterisk(*) next to the {imports,exports} group

[1] $C_i s$ are classes and A is an attribute group

of the first class i.e., EUROPEAN–COUNTRY indicates that we will choose that group as a *seed* for merging with other classes. This seed group is chosen randomly. The next step is to find matching groups for the seed in all other classes. This is done by the find_match() procedure which given a seed and a class returns the largest subset of attributes of the seed that it can find in any group in the class. The results of find_match() for each of the classes in C are shown in the third column. The next step is to find the ratio of the space occupied by the matching groups in the classes of C to the space needed to store the group A for the superclass S. The ratio should be higher than the MERGE-THRESHOLD to allow merging the matching clusters to the cluster A in S . Intuitively this is to ensure that the attributes in A occur sufficiently through the classes in C to justify merging the matching groups to the group A in S. In this example *totalsize* i.e., the space occupied by the matching groups is (2+2+1+0+2+2) = 9 units. The *totalseedsize* i.e., the space that should be occupied in case of an exact merge is 2*6= 12. Thus the ratio is 9/12=0.75 and we do merge to the group {imports,exports} for the superclass COUNTRY (assume that MERGE-THRESHOLD is 0.7). Table 2.2 shows the same set of classes after the merging step when the attributes in A= {imports,exports} have been removed from the classes in C. In case we do not merge the groups we do not remove the attributes from the classes. However we mark the seed group as 'down' so that it is not picked again as a seed. We then pick a seed from the next class ASIAN–COUNTRY and repeat the above procedure.

set of subclasses C with attribute groups	matching groups	size
(EUROPEAN-COUNTRY,{*imports,exports},{area,gdp,economy })	{imports,exports}	2
(ASIAN-COUNTRY,{imports,exports,climate},{debt,economy })	{imports,exports}	2
(AFRICAN-COUNTRY,{imports},{population, languages })	{imports}	1
(N.AMERICAN-COUNTRY,{climate,terrain},{government },{literacy })	{}	0
(S.AMERICAN-COUNTRY,{area, coastline},{imports,exports}	{imports,exports}	2
(AUSTRALIAN-COUNTRY,{imports,exports,debt},{gdp,defense})	{imports,exports}	2

Table 3.3: Merging across classes

set of subclasses C with attribute groups
(EUROPEAN-COUNTRY,{area,gdp,economy })
(ASIAN-COUNTRY,{*climate},{debt,economy })
(AFRICAN-COUNTRY,{population, languages })
(N.AMERICAN-COUNTRY,{climate,terrain},{government },{literacy })
(S.AMERICAN-COUNTRY,{area, coastline}
(AUSTRALIAN-COUNTRY,{debt},{gdp,defense})

Table 3.4: Classes after one merging step

The main motivation behind the steps of merging attribute groups for a single class and also merging classes based on class/subclass relationships is to keep the description of the classes of data extracted as patterns compact. Finally we also keep count of how many queries each class of data supports i.e., how many queries can be answered using that class. From this we can calculate the ratio of supported queries by each class to the total number of queries in the distribution. A class is finally output as a pattern by the CM algorithm only if this ratio is greater than a threshold known as the *query ratio threshold* (q). The query ratio threshold for a class is defined as the ratio of the number of queries that can be answered using this class to the total number of queries in the query distribution.

3.2.2 Language Learned by CM

We present a brief discussion on the language describing the patterns learned by CM. In CM we analyze the queries that are asked of each individual domain concept. To review what we described above, for each domain concept we may further create subclasses (in the ontology of classes of interest) based on constraints in the query. The subclasses are created based on either equality constraints or numeric constraints. We further merge attribute groups in each class and also merge across class coverings. It is easy to see that CM extracts patterns of the form:
SELECT $A_1, A_2, ...A_n$
FROM C

where $A_i s$ are attributes and C is either a domain concept or some subclass of a domain concept (say) where C is the subclass of D such that D satisfies the constraint P. P is a conjunction of predicates that may be equality predicates (numeric or string) or a numeric range predicate.

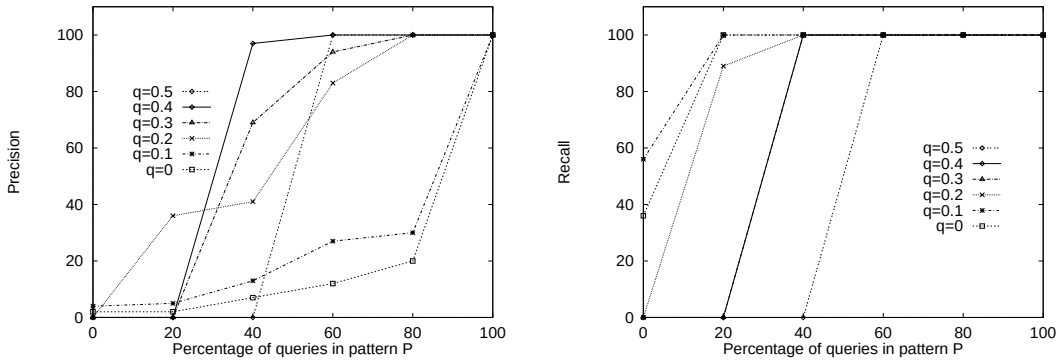
We could extend CM further to learn more general descriptions than at present. For instance one extension would to be able to have more types of constraints in the queries (and the language learned) such as including negation constraints. Also CM currently analyzes queries on single domain classes (or individual subclasses of domain classes). In case of having a large number of join queries across various domain classes it would be interesting to extract patterns that were joins across 2 or more classes (if present in the distribution). One interesting extension to CM would be to also analyze join queries in the distribution and extract join patterns by generalizing the join queries present.

3.3 Evaluating the CM Algorithm

We set up an experiment to evaluate the effectiveness of the CM algorithm in extracting patterns from a query distribution. The experiment is based on standard precision and recall measurements for evaluating information retrieval systems. This is because we are trying to estimate how effective CM is in extracting patterns that are present and also to what extent it extracts extraneous data as patterns.

We first defined a schema for an imaginary mediator application against which we can pose queries. The schema consists of a class S with 50 attributes A1,A2,...,A50. The class S is further partitioned into 5 disjoint subclasses S1,S2,S3,S4 and S5. Each subclass has 10 instances, S1 has instances E1,E2,..,E10, S2 has instances E11,...,E20 etc. We then defined a "pattern" P which is the class S3 with attributes A25,...A30. We then generated different query distributions against this schema varying the percentage of queries that fall within the pattern. We input each distribution to the CM algorithm to see what patterns it would extract from the distribution. We use standard precision and recall measurements from information retrieval to measure the effectiveness of CM in extracting the predefined pattern P. In information retrieval (IR) we specify some criteria for the data we wish to retrieve. For instance we may specify some key words for retrieving text files that contain those keywords (from a large collection of text files). The precision is the percentage of data retrieved that is *relevant*. Data is said to be relevant if it satisfies the search criteria. In this example files that have the keywords specified are said to be relevant and the precision is the percentage of relevant data files in the data files returned by the IR system in response to the search. The recall is the percentage of relevant data retrieved. In this example it is the percentage of the relevant files that were returned by the IR system in response to the search. A good IR system must have both high precision (high precision implies that most of the data retrieved is relevant) and high recall (high recall implies that most of the data that is relevant to our search is indeed returned). In our experiment the predefined pattern P is the *relevant* data, while for each time we run the CM algorithm over a query distribution the patterns extracted from the distribution is the data *retrieved*. It is obvious that we would want a high precision and recall in our system as well. A high precision would imply that a high percentage of the patterns extracted are indeed in the predefined pattern P and a high recall implies that a major portion of the pattern P is identified and extracted by CM. Finally since the query ratio threshold (q) affects what patterns are ultimately output by the CM algorithm we present precision and recall measures for varying threshold values.

Figures 3.4(a) and (b) show the precision and recall values respectively against varying percentages of queries that fall within the predefined pattern P in a query distribution. For each we present precision and recall values for different query ratio thresholds (q). The CM algorithm does indeed prove to be efficient in extracting the predefined pattern P as the recall values are very high (100%) for most of the threshold range for moderate or high percentages of queries in the pattern P. For extraneous data extracted along with P we must analyze the precision values graph. For high threshold values(q=0.4 - 0.5) the precision is very high when a high percentage of queries are in P (> 50%) but very low for lower percentages. This is because even if queries in the pattern P are present, they need to be in a very high proportion for the CM algorithm to extract them at all. For lower threshold (q= 0 - 0.1) the precision is quite low even when a high percentage of queries is in P. This is because of a low threshold the CM algorithm extracts a lot of random classes as patterns in addition to the pattern P. It is best to keep the threshold at an intermediate value (0.2 -0.3) where the precision is high for moderate or high percentage of queries in P. The recall remains quite high (100%) for most of the threshold range for moderate or high percentages of queries in the pattern P.



(a) Precision

(b) Recall

Figure 3.4: Effectiveness of CM Algorithm

3.4 Complexity

As CM has been designed to take as input a large number of previous user queries (a typical number might be say 1000 queries) for extracting patterns, the complexity of running CM is also a matter of concern. We present below a complexity analysis of the algorithm with some reasonable assumptions about the query distribution. For a particular domain

class:
Number of queries = N
Number of attributes of domain class = M
Number of attributes on which constraints may be specified = K

We analyze the complexity by analyzing each step of the CM algorithm:
(i) Creating the ontology of subclasses of interest. For each query
SELECT $A_1, A_2, ...A_n$
FROM C
WHERE P
Suppose P consists of a single predicate. In case of string equality constraints (of the form $A_i = V$) we form subclasses of P based on the value of A_i. We assume that the number of such subclasses is a small constant t. The assumption is valid as we will not form subclasses of C based on an attribute A_i that may be the primary key of C (thus resulting in several subclasses). Also for numeric constraints, both equality and range we partition into subclasses based on ranges for the value of A_i and we can assume that the number of such subclasses is again a small constant t.

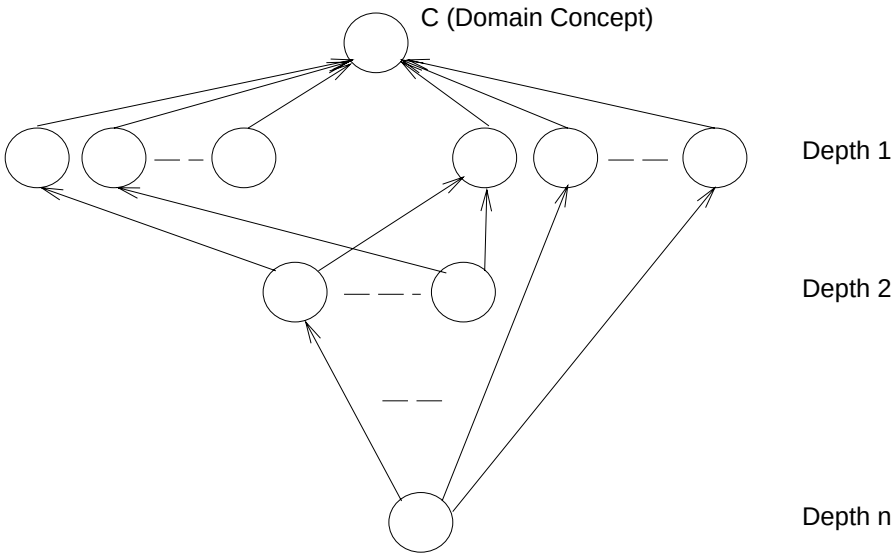


Figure 3.5: Ontology of subclasses of interest

Since we can specify constraints on at most K attributes and the number of subclasses for a single attribute value can be at most t, it is easy to see that in an ontology (see figure 3.5) the number of subclasses is at most $(1+t)^K$. This is because for each of the

K attributes for which we can specify a constraint we may either not specify a constraint or the constraint will be one of the t possible values for the attribute in case of string constraints. In case of numeric constraints it will be in one of the t possible ranges. However for N queries the number of subclasses that can be created will be much less than what $(1+t)^K$ could be. In the ontology at depth 1 we have at most tK subclasses. Now for each query we will create subclasses at depth 1 in the ontology (unless they already exist) and 1 subclass at depth n if the query constraint has n predicates and $N \succ 1$. Thus for N queries the number of subclasses created can be at most $(tk+N)$. For each query we may create upto $(nt+1)$ new subclasses where n is the number of predicates in the query constraint. This is because we could have up to nt new subclasses at depth 1 and 1 new subclass at depth n. For each of these new subclasses we need to ensure that either the subclasses are already present in the ontology or place them at appropriate places in the ontology. A linear search on the ontology is required which takes time at most $(Kt+1)(tK+N)$ for each query as at most K predicates can be present in the query constraint. As the number of queries is N, the entire first step of creating the ontology takes time $N(Kt+1)(Kt+N)$ which is $O(N^2K^2)$

(ii) The second step is for each subclass, to cluster together the groups of attributes based on similarity of cluster and similarity of frequency with which they have been queried. As mentioned earlier we use an approximate 2D clustering algorithm and it has a running time that is quadratic in the number of queries. Assuming an even distribution of the N queries in the $(tk+N)$ subclasses, we have on an average of $N/(tk+N)$ queries in each subclass. Also to measure 'similarity' of attribute clusters in each subclass, we compare clusters pairwise and each comparison takes time at most M^2 as M is the total number of attributes for the domain concept. The clustering of attribute groups in each subclass thus takes time = c $M^2(N/(tk+N))^2$ where c is a constant. As we have at most $(tK+N)$ subclasses the entire clustering step takes time = c (tK+N) $M^2(N/(tk+N))^2$ = c $M^2N^2/(tK+N)$ which is $O(M^2N/K)$

(iii) The final step is to cluster based on coverings. The number of coverings is obviously less than $(tK+N)$, the total number of subclasses in the ontology. Now when we consider a covering we merge across at most t subclasses. This is because we merge across the direct subclasses of a particular class and this can be at most t. Recall that we start with a seed cluster and compare with other clusters to see if there is a good overlap. Each comparison (of the seed cluster with another) takes time M^2. Again assuming an even distribution of queries among the subclasses in the ontology we have an average of $N/(tK+N)$ queries in any subclass. As there are at most t subclasses in a covering, the

total number of attribute clusters is at most $tN/(tK+N)$. If each cluster is compared with every other cluster in the covering (in the course of searching for overlapping clusters), the number of such comparisons is $(tN/(tK+N))^2$. The actual number of comparisons while searching for overlapping clusters to merge will be less than $(tN/(tK+N))^2$ as we will go on removing clusters on merging them. Merging across each covering thus takes time at most $M^2(tn/(tK+N))^2$. As there are at most $(tK+N)$ coverings, the entire third step takes time $M^2(tN)^2/(tK+N)$ which is $O(M^2N/K)$. As the 3 different steps have complexity $O(N^2K^2), O(M^2N/K)$ and $O(M^2N/K)$ respectively, the complexity for the overall algorithm is $O(N^2K^2 + M^2N/K)$. As K can range from 1 to M (number of attributes in the domain concept) this can be further simplified to $O(M^2N^2)$.

$O(M^2N^2)$ seems satisfactory as the complexity is polynomial in the number of attributes in a domain concept and the number of queries analyzed. The time for running CM will thus be acceptable even for large values of M and N. The purpose of analyzing the complexity of CM was to ensure that there are no sources of complexity that would make the running time of CM a matter of concern. For instance if it were say exponential in the number of queries or attributes. However with reasonable assumptions about the query distribution, we have shown that CM is a polynomial time algorithm and thus the running time will not be an issue.

3.5 Summary

We described how the query distribution analysis module performs its task of extracting patterns from user queries. Extracting patterns is important as we want to identify the classes of data frequently queried by users and then materialize them. We presented the CM algorithm that we have developed and use for extracting patterns in a compact manner. We also presented experimental results demonstrating the effectiveness of this algorithm in extracting patterns. Finally we described the language of patterns learned, complexity of the CM algorithm and relationship of extracting patterns with extracting association rules in data mining.

Chapter 4

Source Structure Analysis

We provide database like access to semistructured Web sources through wrappers around the Web sources. Certain kinds of queries to a wrapped Web source can be very expensive as the wrapper might have to perform query processing functions not provided by the Web source. In many cases the processing of such queries can be sped up considerably if some of the data is materialized locally. We have developed an approach for source structure analysis to determine in advance expensive queries to a source. We then prefetch and materialize data that can improve the response time for such queries. Our approach to prefetching data by source structure analysis is the focus of this chapter.

4.1 Introduction

The purpose of source structure analysis is to determine for each mediator application, expensive classes of queries and then prefetch and materialize data that can improve the response time for such expensive queries. Note that we cannot determine classes of data to prefetch by just analyzing the query distribution as described in the previous chapter. By query distribution analysis we determine frequently queried classes of data and materialize them. The complete answer to many user queries is then contained in these materialized classes. By doing source structure analysis we materialize data that is used to speed up the *processing* of expensive classes of queries, as opposed to query distribution analysis where the objective is to materialize data that contains complete *answers* to frequent queries.

Source structure analysis is comprised of the following steps.

- Identifying classes of queries that could be asked in a mediator application.

  We first identify the kinds of queries that can ever be asked in a mediator application. In most mediator applications the user can query the mediator only through a user

interface. The interface is formally described using a GUI specification. We use this specification to determine the kinds of queries that could be asked.

- Identifying expensive queries.

  From the queries that could be asked in a mediator application, we then identify the ones that may be expensive to answer. The cost estimation is done using a query cost estimator in the mediator.

- Selecting the data to materialize

  Finally we identify classes of data that can improve the response time for the expensive queries. This is done using heuristics that are described more in detail later.

We now describe each of the above steps in more detail. First however we provide a brief description of certain aspects of query processing techniques in the SIMS mediator. Some of the information compiled for query processing can actually be used in determining what queries would be asked and also what operations would be performed on data from various sources to answer the queries. As we shall describe later, this information is useful in identifying data that can be prefetched and materialized to speed up answering the expensive queries.

4.1.1 Axioms for Query Processing

In the SIMS system the user poses queries to classes in the domain model and it is the task of the mediator query processor to determine what source or combination of sources must be used to get the answer. In previous work in SIMS [Arens *et al.*, 1996] the selection of sources was performed dynamically by searching the space of query reformulations given the domain model and source descriptions. However the approach did not scale well to large numbers of sources since the search space becomes quite large as the number of sources increases. In our current approach we precompile [Ambite *et al.*, 1998] the source definitions into a set of domain *axioms* which compactly express the possible ways of obtaining the data for each class in the domain model. These axioms can be efficiently instantiated at run time to determine the most appropriate rewriting to answer a query.

Such axioms are illustrated in 4.1. For instance the first axiom states that data for the domain class country can be obtained from the single source Ciafactbook. Axiom 2 states that data for the domain class restaurant can be obtained from either of the two sources, Zagat or Fodors etc. As these axioms explicitly state what sources are needed to obtain the data for a domain class and also what operations will be performed in the data, they

1. country(name area coastline gnp population ...)≡ ciafactbook(name area coastline ...)

2. restaurant(name address telephone) ≡ zagats(name address telephone) OR fodors(name address telephone)

3. restaurant(name address telephone cuisine lat long) ≡ zagats(name area telephone cuisine) AND geocoder($area lat long)

4. jointfaculty(name rank) ≡ csfaculty(name rank) AND businessfaculty(name rank) AND name = name

Figure 4.1: Examples of Axioms

can be exploited to determine possible queries to sources and also determining what data to prefetch and materialize. This will be illustrated in detail in the subsequent sections.

4.2 Identifying Possible Queries

The user queries are posed to classes in the domain or world model. However in most mediator applications the user does not have to explicitly write the queries, rather queries are posed through a graphical user interface (GUI). Typically the user interface places further restrictions on what queries a user can ask. For instance the user may be able to request only certain attributes, be able to specify selection conditions on only some of the attributes, also for a selection condition it may be that the value (of an attribute in a selection condition) must be from a predefined set. To be able to provide the specification of a GUI for a mediator application we have developed a GUI specification language. Having a declarative specification of the GUI enables us to determine what queries can be asked of the application. As mentioned earlier, this may be more restrictive than the kinds of queries we could ask if we directly write structured queries without going through an interface. Another benefit of having a GUI specification is that the GUI can be generated automatically from this specification.

We first provide an informal description of the GUI specification language. There are a number of things that we must specify in order to completely describe a GUI (from the point of view of encoding what queries can be asked using the GUI). First we must specify what are the attributes that can be requested. For instance in the countries application we can request attributes such as area, coastline, national product etc. In this particular application we can choose which of these attributes to retrieve. However there may be GUIs for other applications where the user does not have the choice of selecting what attributes to retrieve. For instance in TheaterLoc, the GUI has been designed such that

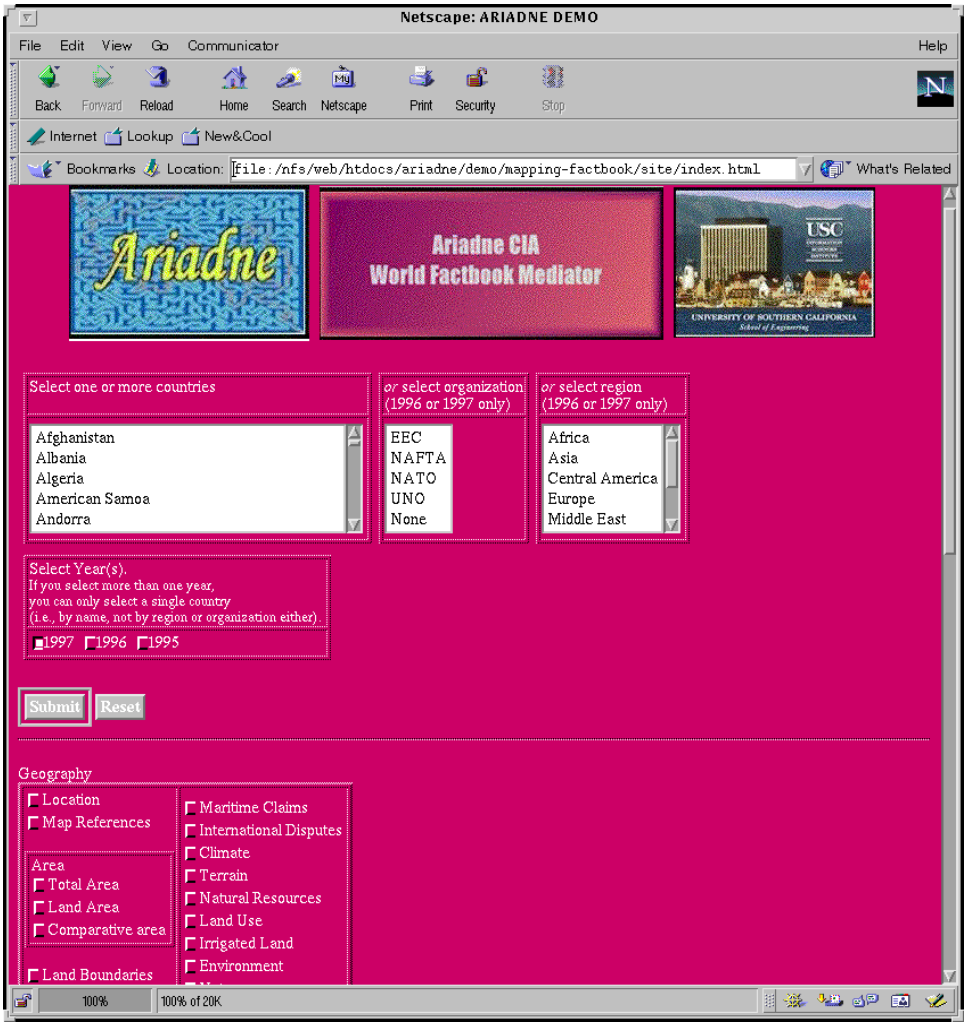


Figure 4.2: Graphical query interface for countries mediator

the user always gets back the name, address and latitude and longitude of restaurants and theatres in a selected city. The GUI specification must thus also reflect whether or not the user has a choice of what attributes can be retrieved. Next we must specify what kind of selection constraints the user can specify using the interface. For instance in the countries application the user can specify constraints on either of name, region or organization attributes. Also for each of the above attributes the user may select a value for that attribute from one of a prespecified list of values. For instance region must be assigned to one of Asia, Europe, ... etc. We must also specify whether it is necessary that the user specify a constraint on such attributes. For instance in the countries application it is optional that the user specify a constraint on the name, region or organization attributes.

However the TheaterLoc GUI is such that the user must always specify a city in which he wishes to locate theatres and restaurants.

We use an SQL like syntax for the GUI specification. In the SELECT clause we specify what attributes can be retrieved and also whether the user has a choice of specifying what attributes to retrieve. In the WHERE clause we specify what attributes the user can specify constraints on, for each such attribute whether it is necessary to specify the constraint and also whether the attribute value must be assigned to one of a fixed set of values. In that case we also list the set of these values. As an example consider the GUI specification for the countries mediator shown in 4.3. In the SELECT clause we specify the information that the attributes area, coastline,..., economy can be retrieved. We also specify (using '' braces) that retrieving each attribute is optional. In the WHERE clause we specify information such as the user can specify constraints on attributes like name (the value of which must be one of Albania, ...,Zambia) etc. As another example consider the GUI specification for TheaterLoc shown in 4.4. The SELECT clause specifies that we retrieve attributes name, address, latitude, longitudes and maps of places. Also the set of attributes retrieved is always fixed (shown by the '[' brace). The WHERE clause specifies that the user must specify a selection constraints on city, the value must be one of the given set of values.

A BNF for the interface specification is provided in 4.5.

```
SELECT {area, coastline, ... economy}
FROM countries
WHERE {name,1,(Albania, ...)} {region,1,(Asia,...)} {organization,1,(EEC,..)}
```

Figure 4.3: Example of Interface Specification

```
SELECT [name,address,latitude,longitude,map]
FROM places
WHERE [city,1,(Los Angeles, Santa Monica, ... )]
```

Figure 4.4: Example of Interface Specification

4.3 Axiom Pruning

Given the domain model and descriptions of information sources in terms of the domain model the system precompiles axioms for all classes in the domain model. However as we

```
<interface>  ::=  SELECT <fixedprojectlist> <optprojectlist>
FROM <application>
WHERE <conditions>
;
<conditions>  ::=  <cond> | <cond> <conditions>
;
<cond>  ::=  <freecond> | <fixedcond>
;
<freecond> ::= [ <attval> ] | { <attval> }
;
<fixedcond>
::= [ <attval> , <num> , <range> ] | { <attval> , <num> , <range> }
;
<range> ::= ( <valueslist> )
;
<valueslist> ::= | <valueslist> <attval> | <valueslist> , <attval>
  ;
<fixedprojectlist>  ::= | { <valueslist> }
;
<optprojectlist>  ::= | [ valueslist ]
;
<attval>  : ATTVAL
;
<num> : NUMBER
;
<application> : APP
;
```

Figure 4.5: BNF for the interface specification

described earlier, the user interface can place further restrictions on what queries can be asked by users. As a result, not all of the axioms generated will be *relevant* i.e., be used when answering queries in an application. For instance consider an interface specification where the user always asks for all the following attributes of restaurant (name address telephone lat long). In this case axiom 2 is not relevant as it does not provide the attributes lat and long. However axiom 3 is relevant as it provides all the attributes. Further if in the interface we always had a selection condition on the cuisine attribute then axiom 2 would not be relevant as the cuisine attribute is not provided in the domain class restaurant. However axiom 3 would be relevant.

```
PRUNE_AXIOMS (S) {
FOR (all axioms A in S) {
      D = domain_concept(A)
      M = mandatory_projection_attributes(I)
      O = optional_projection_attributes(I)
      C = mandatory_condition_attributes(I)
      IF(all_present(M,D) and one_present(O,D) and all_present(C,D) {
        is_relevant(A) = TRUE
      }
      ELSE {
         is_relevant(A) = FALSE
      }
}

all_present(X,D) {
return TRUE if all attributes in set X present in attributes of domain class D
return FALSE otherwise
}

one_present(X,D) {
return TRUE if at least one attribute in set X present in attributes of domain class D
return FALSE otherwise
}
```

Figure 4.6: Algorithm for determining relevant axioms given a user interface specification

The algorithm for pruning axioms is given in figure 4.6. It is a straightforward algorithm which marks an axiom as relevant (to a given interface) after checking the following. First all the attributes that are necessarily retrieved in a user query are present in the domain class in the axiom. Also all the attributes for which a selection condition must (necessarily) be specified are also present in the axiom domain class.

4.4 Heuristics for Prefetching Data

Finally we analyze all the relevant axioms (in conjunction with the interface specification) to determine possibly expensive queries. Based on the kind of query (i.e., selection query, join query etc.) and the database operations that are performed on a source or group of sources we can determine classes of data which if prefetched and materialized will improve the response time for the query. We enlist heuristics that tell us what classes of data to prefetch for each kind of database operation that can be performed on the sources.

Unary operations:

1. The GUI specification states that only projections are allowed on the attributes in the source relation

For instance consider an axiom such as axiom 1 above i.e.,

country(name area coastline gnp population ...) $\equiv$ ciafactbook(c.name c.area c.coastline ...)

and suppose the query interface is such that we can only project out different attributes of the country domain class. In this case the only query that can be asked of the source class is to retrieve one or more attributes for all tuples in the source class. It is best to analyze the query distribution to see if the domain class is queried frequently and also what groups of attributes are queried frequently.

2. The GUI specification states that we can specify constraints on certain attributes in the domain class (i.e., selection queries allowed)

Consider again axiom 1 but suppose the query interface is such that we can specify selection conditions on some attributes, say on attributes region and organization. A selection operation can be expensive if it is not supported by the source. In that case it involves scanning over all the tuples i.e., retrieving pages of all tuples from the Web source. The ciafactbook is such a source where any selection operation involves fetching pages of all 270 countries from the Web source which is very time consuming. In case the number of attributes on which selection conditions can be specified is 'small'[1] we materialize the primary key and the the attributes of the source relation on which selection conditions

[1]The number of attributes is less than some preset threshold value in the system

can be specified. This enables us to perform the selection operations locally and thereby avoid having to fetch all tuples (pages) of the source relation and the query response time is considerably improved.

Binary operations:

3. Unions between source relations

For instance consider axioms such as axiom 2 i.e.,

restaurant(name address telephone) $\equiv$ zagats(name address telephone) OR fodors(name address telephone)

For such queries both source relations can be just analyzed independently. The fact that we are doing a union operation over the relations does not in any way affect the costs of queries to the individual sources. Thus in this case we will not prefetch any data.

4. When we have joins between two source relations

If the join is expensive then we should materialize the primary keys of both sources relations as well as the attributes of the relations on which the join is being performed. Then the join can be performed locally which improves the query response time.

5. When we have ordered joins between two source relations (in case of binding pattern constraints on attributes).

An ordered join is a special kind of join across Web information sources. An ordered join is performed in case a join has to performed across sources one of which has a *binding pattern* constraint. A binding pattern constraint essentially states that the value of one or more attributes in the relation must be bound to get any data at all. For instance in the above axiom 4 there is a binding pattern constraint on the address attribute in the geocoder implying that the address must be supplied to get any data (in this case the latitude and longitude of that address) from the geocoder source. To get the latitudes and longitudes for the restaurants we first retrieve the addresses of all restaurants (from the Zagatsla source) and then geocode them one after another. Due to the binding pattern constraint that the address has to be provided and the structure of the online geocoder which is such that it accepts only one address at a time, we can only geocode one address at a time and this is exactly why ordered join operations are generally expensive.

There are 2 factors in deciding what to prefetch and materialize. First we must identify on what data the ordered join(s) can be performed. For instance for an axiom such as 3 and with an interface where no selection conditions can be specified on any of the attributes the only data on which the ordered join can be performed is to geocode all addresses from restaurants in the zagats source. However suppose we have an interface where we must specify the cuisine type and also the only kinds of cuisine we can select are

chinese or mexican. In that case the only ordered joins that would be performed are to geocode addresses of either chinese or mexican restaurants from the zagats source. There is a straightforward algorithm to determine on exactly what data the ordered join can performed, given the axiom and the details of the selection conditions in the interface specification and we do not describe it further here.

Next we must determine what attributes are to be prefetched and materialized. In case the domain class has a 'small' number of attributes (such as in axiom 2) we simply materialize all the attributes in the domain class. However if the number of attributes is very large (such as in axiom 1) we materialize just the primary key and the attributes in the domain class on which selection conditions can be specified.

6. Set Difference For instance consider an axiom such as:

foreign_students(name,major) ≡ students(name,major) - resident_students(name,major)

It may be the case that both the source relations are very large, but the difference is very small. For instance in the above example it may well be that the number of students is large and also that almost all students are indeed resident students. In that case the number of foreign students would be small and we could materialize this class.

7. Cartesian Product

The cartesian product of two relations generally results in a very large relation. We should thus wait to analyze the query distribution before materializing a cartesian product of two classes..

8. Set Intersection Consider an axiom such as:

double_majors(name,age) ≡ cs_majors(name,age) and ee_majors(name,age)

In many cases the two source relations could be very large relations but the intersection will be a small relation that we can materialize. As in the case of set difference we materialize the intersection if it is small.

9. The Division Operation

s1(a b) div s2(a)

Division is another time consuming operation. There will be cases where both s1 and s2 are large relations but the relation obtained after division is very small. Again as in the case of the difference and intersection operations above we should materialize the resulting relation.

4.5 Materializing the Classes

Source structure analysis proposes a set of classes to prefetch and materialize as described above. However we need to analyze some other factors as well before simply materializing these classes. One factor we need to take into account is the maintenance cost for these classes in the face of updates at the original sources. By analyzing the distribution of queries using CM we will also have other classes that we must consider materializing. Besides for each mediator application we assume we have a fixed amount of local space to store the materialized classes and a fixed amount of maintenance cost that can be borne to keep the materialized data up to date. Because of these constraints it may not be possible to materialize all the classes proposed by source structure analysis or CM and we must make an optimal selection of these classes. The factors considered and the process of ultimately deciding to materialize a proposed class (on the basis of source structure analysis or CM) are described in detail in Chapter 6. There we describe how exactly multiple factors are considered in combination for materializing classes of data and an admission and replacement policy for the materialized data store.

4.6 Summary

Let us summarize what we presented in this chapter. We introduced the idea of source structure analysis which is a process of analyzing the structure of sources to predetermine expensive kinds of queries to the sources and prefetching and materializing data to improve response time for these queries. We described how precompiled axioms are used in query processing in SIMS. We described a GUI specification language that captures exactly the kinds of queries that can be asked through a particular user interface. We showed how the user interface specification is used to identify the kinds of queries that can be asked and the axioms that would be used. Finally we showed how using the interface specifications, and the axioms that specify the sources used and operations performed on the data, we are able to identify classes of data to materialize that can improve response time for the expensive queries. The decision of actually materializing such classes is made considering other factors and is discussed in more detail in Chapter 6.

Chapter 5

Updates at Sources

We provide integrated access to data from Web sources and the data at these sources may change. This can cause the materialized data to be inconsistent with that in the original sources. In this chapter we discuss our approach to handling the issue of updates and changes at the Web sources from which we materialize data.

5.1 Approach to Updates

The data materialized may change at the original Web sources. We must also provide the user with consistent data even if it is accessed from the materialized data store. This impacts our decision of materializing a particular class of data. In fact in some cases the data may be updated so frequently that we should not consider materializing it at all. As an example consider an online information source that provides the latest information on flight arrivals and departures at a particular airport. Assume that the source is updated every 10 seconds. Also it is fairly obvious that any user would want the most recent data from this source. In case we were to materialize data from this source it would have to be updated every 10 seconds which leads to a very heavy maintenance load for the mediator. In such a case we may decide not to materialize data from this source at all as it is very frequently updated.

For classes of data that we do materialize, our strategy is to refresh the materialized data when it changes. There are two issues that then need to be addressed. First if the class of data is indeed materialized then what is the frequency with which we need to refresh the materialized data. Also, associated with each materialized class will be a *maintenance cost* which is essentially an estimate of the time spent to keep the materialized class up to date. It depends on the update frequency and the cost of querying the class using the original sources and will be discussed more in detail later. The other issue is how does the

maintenance cost affect the decision of whether or not to materialize a particular class. Note that for any source we select portions of data to materialize based on the update frequency and cost of various attributes. For instance for an information about movies source we may materialize attributes such as movie listings, address and telephone number of theater etc. that changes once a week (at most) and decide not to materialize data such as 'additional screenings' (last minute additional shows for some very popular movie) which could change anytime. Another important feature of our approach to updates is that we assume that the user will not necessarily always want the most recent data. He may in many cases be willing to accept data that is a little stale in exchange for the advantage of retrieving it fast. For instance there may be an online source providing brief descriptions of cities in the US from the perspective of each city as a place to live, and it may be that the source is updated every week. However it may be that the user is willing to accept information about a city which is a month old.

For some sources the update time and frequency is known in advance. For instance for the `Yahoo movies` source we know that it is updated once a week, every Friday. Other sources may change at any time. For instance information at the `Zagat` source may change any time. We should mention that there has been work done on automatically detecting changes at Web sources [Chawathe *et al.*, 1996] and a direction of future work is to incorporate change detection into the update aspect of our system. For now we shall simply refresh the classes of data materialized from sources that can change any time at the frequency and time determined purely by user requirements.

To provide an outline of our approach to updates, we first specify for each mediator application the update characteristics and frequencies of various sources and attributes. We also specify the user's requirements for freshness of data. We then estimate, for each class of data we consider materializing, the maintenance cost for that class of data. The maintenance cost is then factored into the decision of whether or not to materialize a class of data. In this chapter however we will concern ourselves only with the issues of how we specify the update characteristics of a source and also how we compute the maintenance cost for a class of data. How we factor the maintenance cost into the decision of whether or not to materialize a class is to be considered in the context of the entire materialization system and we will leave it for the following Chapter.

5.2 Source Update Specifications

I have developed a language for describing the update characteristics and frequency of updates for various source classes and attributes and also the user's requirements for freshness of each domain class and attributes. I will illustrate the kinds of characteristics we can specify using an example. Consider an information source for movies which we model as a source relation MOVIE_SRC having attributes such as theatre, showtimes, actors, director, review etc. Table 5.2 shows the update specification for the movies relation and some attributes, the specification is implemented as a database relation.

CLASS	MEMBERSHIP	CHANGE	TIME_PERIOD	TIME
MOVIE_SRC	A	Y	1 week	week:friday

Table 5.1: Characteristics of updates for a source class

It states that for the MOVIE_SRC source class, the MEMBERSHIP='A' i.e., arbitrary (instances of the class can be added or deleted), CHANGE = 'Y' i.e, yes (values of objects or class members can change), TIME_PERIOD is 1 week so the data changes once every week and the TIME of change is every friday. The attributes CHANGE, TIME_PERIOD and TIME also characterize each attribute of a source class. By default for each attribute in a source class the values of the update characterization attributes are propagated from the source relation they are part of. So for instance the attribute 'theatre' has the characteristics CHANGE='Y', TIME_PERIOD= 1 week etc. These defaults can be overridden by explicitly stating the new values. So for instance we could specify the characteristics for the 'actors' attribute as shown in Table 5.2 since the value of actors for MOVIE_SRC does not change.

ATTRIBUTE	CHANGE	TIME_PERIOD	TIME
actors	N	-	-

Table 5.2: Characteristics of updates for an attribute

More formally, update specifications are made for source classes and (optionally) for attributes of source classes. The update specification is the database relation shown in Table 5.2. $< class >$ is the name of the source class. $< mem >$ standing for object membership in a class is one of 'I','D','A'. 'I' implies the membership of objects is monotonically increasing. For instance suppose we have a class that consists of all the actresses that have ever won an Oscar for best actress, then obviously such a class is monotonically increasing. 'D' implies that the membership of objects in the class is monotonically decreasing. An example of such a class could be a class of personnel that served in Operation Desert Storm and are still in military service. 'A' implies arbitrary membership. For instance

consider a class of USC faculty that are NSF Young Investigators, objects to this class may both be added or deleted. This membership information is useful in that it further helps the materialization system to optimally respond to changes to data in these classes. For instance for a monotonically increasing class that we have materialized, we know that the materialized data is valid for good and that objects to this class may only be further added and not deleted. $< change >$ is one of 'Y', 'N' depending on whether the source classes changes or not. $< period >$ is the time period of update if the source changes and $< time >$ is the time (day of week, month of year etc.) when the source will change.

CLASS	MEMBERSHIP	CHANGE	TIME_PERIOD	TIME
$< class >$	$< mem >$	$< change >$	$< period >$	$< time >$

Table 5.3: Update specifications for a source class

ATTRIBUTE	CHANGE	TIME_PERIOD	TIME
$< attribute >$	$< change >$	$< period >$	$< time >$

Table 5.4: Update specifications for source attributes

Table 5.4 shows the update specification relation for attributes. $< attributes >$ is a source attribute and $< change >$, $< period >$ and $< time >$ have the same semantics as for the source specifications.

For each domain class we also specify for each attribute the user's requirements for freshness of data, actually stated in terms of how stale he can tolerate the data to be. Consider a domain class called MOVIE where we integrate information from several sources about movies.

ATTRIBUTE	TOLERANCE
theatre	0
showtimes	0
actors	0
director	0
review	6 weeks

Table 5.5: Staleness tolerances for attributes of a domain class

Table 5.5 shows the user's tolerance for various attributes of the domain class MOVIE. It states that the value of attributes such as 'theatre' and 'showtimes' must be current (having a tolerance of 0) whereas the 'review' can be up to 6 weeks old. Formally, a domain class user requirement is specified as the database relation in Table 5.6 where $< class >$ is a domain class and $< tolerance >$ is the user's tolerance for staleness of data from

that class. Table 5.7 shows the attribute tolerance specifications where $< attribute >$ is a domain attribute and $< tolerance >$ is the user's tolerance for staleness of that attribute.

CLASS	TOLERANCE
$< class >$	$< tolerance >$

Table 5.6: Staleness tolerance specifications for a domain class

CLASS	TOLERANCE
$< attribute >$	$< tolerance >$

Table 5.7: Staleness tolerance specifications for attributes of a domain class

```
READ_UPDATE_SPECIFICATIONS() {
read_sourceclass_update_specifications()
/* read update specifications for source classes */
propagate_source_update_specifications()
/* source class attributes inherit from source class */
read_domainclass_tolerance_specifications()
/* read domain class tolerances */
propagate_domain_tolerance_specifications()
/* domain attribute tolerances inherit from domain class */
read_sourceattribute_update_specifications()
/* read any explictly specified source attribute specifications */
read_domainattribute_update_tolerances()
/* read any explicitly specified domain attribute specifications */
}
```

Figure 5.1: Reading Update Specifications

The update specifications are specified in the language described above. The procedure READ_UPDATE_SPECIFICATIONS shown in Figure 5.1 shows how update specifications are read into the materialization system. Specifications for each source class are specified explicitly and by default propagated to the attributes of each source class. The tolerance specifications for each domain class are also specified explicitly and propagated by default to the attributes of each domain class. We may also specify explicitly update characteristics of individual attributes of source classes or tolerances of attributes of domain class.

5.2.1 Determining Maintenance Cost

The maintenance cost for a materialized class is one factor that must be taken into account before materializing it. We define the maintenance cost M for each class to be the time

spent per day for retrieving the data for that class from the original sources to refresh the materialized class. Just as we have a limited local space to store all the materialized classes, we assume that we also have a limited total maintenance cost that can be borne by the mediator for keeping all the materialized classes up to date. For each class of data that we propose to materialize we must thus make an estimate of what will be the maintenance cost for the materialized class. The maintenance cost is a product of the time taken to query the class using the remote sources and the number of times it is updated per day. Thus we have: $M = F * C$ where F is the update frequency(number of updates per day) and C is the time to query the class using the remote sources. C is estimated using a query cost estimator in the Ariadne system. For determining F, the update frequency of a class, we have an extremely simple procedure. Consider that the proposed materialized class has attributes $A_1, A_2, ..., A_n$. For each attribute A_i in the proposed materialized class we look up the tolerance O_i for that attribute from the user specifications for the domain class and attribute tolerances. From the source definition we determine what attribute in a source class maps to the (domain class) attribute in the proposed materialized class. We look up the update time period P_i of that source attribute. The time period T_i with which the attribute in the proposed materialized class must be updated is simply the greater of O_i and P_i. $T_i = max(O_i, P_i)$ The update frequency for each attribute $F_i = 1/T_i$. The update frequency for the entire class which we propose to materialize is taken to be the maximum of the update frequencies of the individual attributes. $F = max(F_i)$. The procedures for determining maintenance frequency and cost are shown in Figure 5.2

5.3 Predictable and Unpredictable Changes

Our approach is based on the assumption that the time and frequency of updates at any source is known. This holds true for some sources. For instance we may know for sure that data at a quote server source is updated every second starting at 8:00 am and up to 5:00 pm each day. But consider a source such as a Web site providing information about the faculty in computer science at some university. Such data can be updated anytime. In our approach we make a reasonable guess of the update frequency say we assume it is updated once every month. Techniques have been developed for change detection of semi-structured data [Chawathe *et al.*, 1996] with a particular focus on semi-structured data in Web sources. In future we intend to incorporate a subsystem for detecting changes at sources for which the time of change is not known in advance and then notifying the materialization system of such changes.

Given: Proposed materialized class M with attributes A=$A_1, A_2, ..., A_n$
Returns: Frequency with which M should be refreshed
MAINTENANCE_FREQUENCY(M, A) {
n=size(A)
for (i=1 to n) do {
  O_i = tolerance(A_i)
  B_i = mapsto(A_i) /* what source attribute maps to A_i */
  P_i = updatetimeperiod(B_i)
  T_i = max(O_i, P_i)
  F_i = 1/(T_i)
}
F=max(F_i)
return(F)
}

Given: Proposed materialized class M with attributes A=$A_1, A_2, ..., A_n$
Returns: Maintenance cost if we materialize M
MAINTENANCE_COST(M, A) {
F=MAINTENANCE_FREQUENCY(M, A)
C = query_cost(M, A)
return($F * C$)
}

Figure 5.2: Procedures for determining maintenance frequency and cost

5.4 Impact of Updates on Joins

There may be situations when some of the data we have materialized is not consistent with that in the original store, specifically in cases where the user has requirements when he is willing to accept data that may not be the most recent. We then use this materialized data when the user queries this particular class. However there could be problems when performing operations such as a join between this materialized data that is stale and fresh data from the original sources. In this case we will end up performing joins between different versions of data which can lead to incorrect answers. As an example consider a case when we have materialized a class containing data about the computer science faculty at some university. Assume the source from which we get this data may change anytime however the user is willing to accept data that is up to 1 month old. Now the data we have materialized in this class may thus be as much as 1 month old. Now consider a query which asks for faculty members that have joint appointments in computer science and electrical engineering departments. This query involves performing a join between the computer science and electrical engineering classes. Now we could do the join using the materialized class for computer science faculty (which may be 1 month old) and the electrical engineering faculty from the original Web source which is current. The answer really is now "people who were faculty members in computer science one month ago and who are faculty members in electrical engineering now". This is neither equivalent to faculty members who had joint appointments one month ago (which is an answer that the user may be willing to accept) or do so presently (which is exactly what the user requested).

There are alternative ways to address this problem. One strategy is to ensure that classes of data across which joins may be performed are always up to date (regardless of the user's tolerances). We thus refresh these classes at the frequency with which the source or attributes are updated. Determining what classes of data joins may be performed across is straightforward as the query processing axioms have this information. An alternative strategy is that join queries across domain classes are always answered using the original sources instead of any materialized data. We have gone with the former approach. The advantage of the former approach is that data is materialized so responses to queries involving this data will of course be faster. This also means that the maintenance cost may increase as the materialized classes must now be always kept up to date. The latter approach of course leads to a decline in performance as the queries involving joins are now answered using the remote sources.

5.5 Summary

We introduced our approach to handling updates at sources. Our goal is to provide the user with consistent data (according to his requirements). We presented a language for describing update characteristics of sources and also user requirements for freshness. We presented algorithms for estimating the maintenance frequency and cost of a given class given such specifications. We also pointed out the impact of updates on joins involving materialized data and data from the original sources and our approach to the problem. In the next chapter we will see how the maintenance frequency and cost are factored into the decision of materializing a class of data.

Chapter 6

The Integrated Materialization System

We now describe how exactly various modules of the materialization system are pieced together to perform the overall materialization task. We already discussed the order of analysis and data flow between modules in Chapter 2. In this chapter we discuss the admission and replacement policy for classes of materialized data and also the tasks of the 'main' materialization module. In essence we will conclude the description of how multiple factors are combined to perform the materialization task for performance optimization.

6.1 Admission and Replacement

Classes of data to materialize are proposed by both query distribution and source structure analysis. As we mentioned earlier, we may not be able to materialize all the classes as we probably will have a limited space for storing the materialized data and also a limited maintenance cost that we can bear for keeping the materialized classes up to date in the presence of changes. The question is then how do we optimally utilize the given space and maintenance cost for storing the materialized classes of data. Clearly if we cannot materialize all classes, we must materialize those that will most reduce the query response time for the mediator. Let us formalize the problem. For each proposed materialized class we have an estimate of how much savings in query response time we get by materializing it (we discuss the savings estimation in the following subsection). Also each class takes a certain amount of space and has a certain maintenance cost associated with it. The problem is then given a limited total space and total maintenance cost, what subset of classes must we choose so as to maximize the total savings in query response time. We present our solution after first describing how exactly we estimate the savings in query response time for each proposed materialized class.

6.1.1 Estimating Query Response Time Savings

The classes proposed for materialization are either on the basis of query distribution or source structure analysis. When we materialize classes output by query distribution analysis, future queries to those classes are answered by retrieving all the data from a local database as opposed to the remote sources. We consider the time to retrieve the data from the local database to be negligible as compared to retrieving it from the remote sources. Thus the *savings* achieved in query response time is simply the time taken to answer all the queries in the query distribution that fall within the materialized class, from the remote sources. Recall that query distribution analysis proposes classes to materialize by merging several classes (queries) in the query distribution. For each such query, the query cost estimator gives us an estimate of the query response time using the remote sources. Initially i.e., before we run the CM algorithm, this cost estimate is taken to be the savings for each class (query). In CM, as we merge classes, the savings for a merged class is taken to be the sum of the savings of all classes that are merged to form the merged class.

For the classes proposed by source structure analysis we again consider the savings time to simply be the time taken to answer the queries that could be answered using the materialized class, from the remote sources instead. The estimation is a little more complex as compared to estimating the savings for classes proposed by query distribution analysis. First for each class we prefetch, we must estimate the time to answer the query that we intend to improve response time for from the remote sources. Consider the case of materializing the primary key and selection attribute for a source for which selection queries on that attribute are expensive. Using the cost estimator we get an estimate of the query answering cost using only the remote source(s). However the *savings* for the class we prefetch is the *total* savings in query response time due to this class. We thus need to know how many queries in the query distribution use the prefetched class and multiply it by the time taken to answer a single query. The number of queries is not known of course until we analyze at least one user query distribution. For prefetched classes thus the savings is taken to be the product of the query answering time for queries (that will use the prefetched class) using the remote sources and the number of queries in the query distribution that use the prefetched class.

6.1.2 Algorithm for Admission and Replacement

After analyzing the query distribution we have an estimate of the savings for each proposed materialized class (whether proposed by source structure or query distribution analysis).

Let us formalize the problem of how to optimally use the resources of space and maintenance cost for materialization.

Given: A set of proposed classes of data to materialize $C_1, C_2, ...C_m$

Each class C_i has savings V_i

Each class C_i occupies space S_i and has maintenance cost M_i

Total space available for materialized classes = S

Total maintenance cost = M

Find: The subset $C_{j1}, C_{j2}, ..., C_{jm}$ of $C_1, C_2, ..., C_m$ such that $\Sigma S_{ji} \leq S, \Sigma M_{ji} \leq M$ and for which ΣV_{ji} is maximized.

Solution: The above problem is basically the familiar Knapsack problem in 2-dimensions. A greedy algorithm is used for the *fractional* Knapsack problem in one dimension. I present a solution on similar lines for the 2-D (fractional) case. Assume $S = M$ (if not originally so, the $S_i s$ and S or M_i s and M can be scaled to ensure this). The following ensures an optimal selection of classes:

(1) Sort the classes in descending order of the profit $d_i = V_i / max(S_i, M_i)$

(2) Materialize classes in this order till either of S or M is used up.

Proof: The proof that the above greedy algorithm yields an optimal selection of classes to materialize is based on an informal notion of savings "density" i.e., the savings per unit space or maintenance cost. For ease of visualization consider a graphical version of the problem. Consider each class C_i as a vector (x,y) in 2 dimensions, where x=S_i and y = M_i as shown in Figure 6.1. The problem is then to choose vectors that can be fitted subsequently (end of one vector is beginning of next) in the space defined by (S,M) such the total savings (of vectors in this space) is maximized. See Figure 6.2

At each stage we choose (from among the remaining vectors) one with maximum $d_i = V_i / max(S_i, M_i)$. We claim that in space $s = max(S_i, M_i)$ and maintenance cost $m = max(S_i, M_i), V_i$ is maximum savings that can be accommodated. Suppose this were not true i.e., there may exist vectors $C_k 1, C_k 2, ..., C_k n, n \geq 1$ such that $\Sigma V_k j > V_i$. This implies that $\Sigma d_k j * max(S_k i, M_k i) > V_i$. In that case we also have $max(d_k j) * max(S_i, M_i) > V_i$ Thus $max(d_k j) > d_i$

But this is not possible as C_i has been chosen such that d_i is the maximum from amongst that of the remaining vectors. So V_i the savings of the vector with maximum d_i is the maximum savings that can be achieved in space $s = max(S_i, M_i)$ and maintenance cost $m = max(S_i, M_i)$. It follows that each step we use the maintenance cost and space

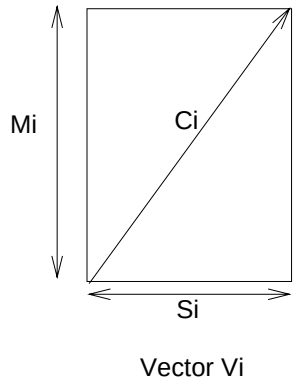


Figure 6.1: Vector Representation of a Proposed Materialized Class

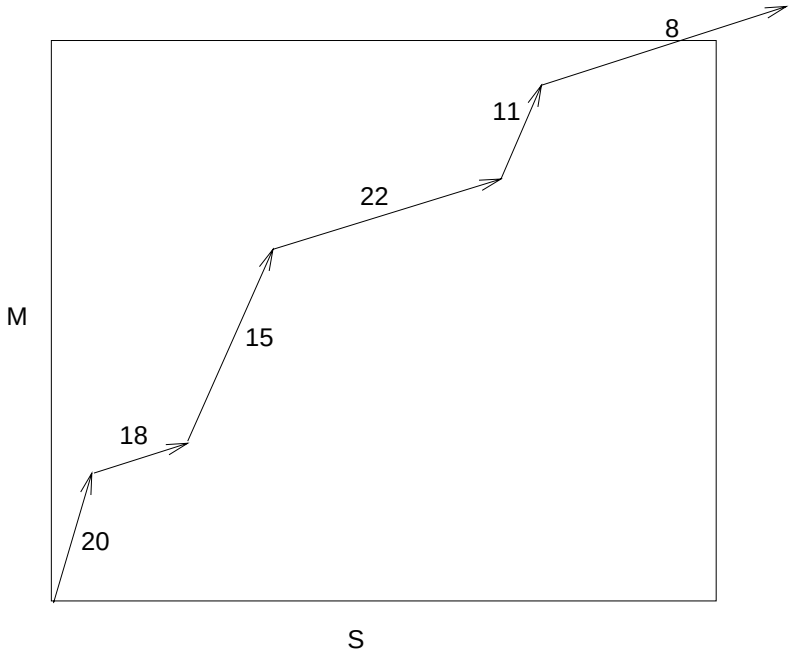


Figure 6.2: Vector Representation of Optimization Problem (2-D Knapsack)

optimally and choosing the vector with maximum d_i is better than choosing any of the remaining vectors. Thus storing the vectors in descending order of d_i does indeed provide an optimal selection of vectors.

The above algorithm thus provides the admission and replacement module with a means to optimally select classes of data to materialize given system constraints of space and maintenance cost. Each class is assigned a profit which is basically the ration of estimated savings of that class to the greater of space occupied and maintenance cost for that class. Classes are materialized in descending order of this profit. The analysis of all the various factors is now complete with the identification of classes of data to finally materialize. The admission and replacement module sends this information to the materialization module which then fetches and materializes this data. We now describe the tasks of the materialization module.

6.2 Materialization Module

The materialization module should be viewed as the 'main' system module. When given a set of classes to materialize (by the admission and replacement module) it issues queries to the original sources and materializes the data. Materializing the data involves populating the local database and also making appropriate changes to the domain and source models for the mediator application to reflect the incorporation of new (materialized) sources. It must also now recompile the query processing axioms as the domain and source models have changed. Finally it must refresh the materialized classes at appropriate time intervals. It gets the frequency with which materialized classes should be refreshed from the update module

Another task of the materialization module is to periodically reevaluate the materialized data. For any mediator application it is quite likely that that user query patterns will change with time. In the materialization system we thus periodically reevaluate the patterns extracted from the user queries by rerunning CM over the most recent queries. We also re-evaluate the frequency with which various classes proposed by source structure analysis are queried. The materialized data store is then re-populated with classes of data that are optimal to materialize in the context of the most recent query distribution. An appropriate time period with which the query distribution must be reevaluated will vary from application to application. For instance for the countries mediator we could reevaluate the materialized classes once a month but for a mediator about theatres and movies

such as TheaterLoc, it will be more appropriate to reevaluate the materialized data at least once every week as movies change every week.

6.3 Summary

We described how the various modules are finally integrated to perform the overall materialization task. We described the admission and replacement policy for materialized classes and also the tasks of the main materialization in interacting with a mediator to optimize performance by materialization.

Chapter 7

Experimental Results

We present experimental results demonstrating the effectiveness of our approach to optimizing performance by materializing data. In Chapter 3 we had presented results to demonstrate the effectiveness of the CM algorithm in accurately extracting patterns from user queries. In this section we demonstrate the effectiveness of a variety of aspects of the materialization system. We demonstrate the effectiveness of source structure analysis, extracting patterns and update cost analysis individually and also considering all the above factors in combination. We also demonstrate the effectiveness of the admission and replacement policy for materialized data classes as query patterns evolve over time.

7.1 Introduction

We provide experimental results that demonstrate the effectiveness of several aspects of our approach to optimizing performance by selectively materializing data. We first provide a brief description of the implementation of the materialization system that we developed for Ariadne based on the architecture presented in Chapter 2. We then outline the hypotheses that we test in the various experiments. We describe 3 different Ariadne applications that we use for the experiments. Finally we present the actual experimental results with discussions.

7.1.1 Implementation of the Materialization System

We have implemented a materialization system for Ariadne based on the architecture described in Chapter 2. The individual modules for source structure analysis, query distribution analysis, updates, admission and replacement, and the optimizer module are all implemented in C. The module for query distribution analysis also uses Powerloom [MacGregor *et al.*,] as the knowledge representation system for maintaining the ontology of

classes of interest identified in a user query distribution. We used the Informix Universal Server Version 9.1 [inf, March1997] as the database system to store and manage the locally materialized data. Also KQML [Finin *et al.*, in press] was used for interprocess communication. The materialization system (including the local Informix database) was run on a Sun Ultra 1 (running Solaris) for obtaining the experimental results.

7.2 Experimental Hypotheses

We have made claims about how several different aspects of our materialization system will be effective in optimizing the performance of a mediator application. We list these various hypotheses explicitly and our experiments are focussed on validating these hypotheses. They are the following:

- Prefetching and materializing data by source structure analysis improves the response time of queries for an application.
- Materializing frequently accessed classes of data extracted as patterns from user queries improves the response time for queries in an application.
- The performance improvement in terms of improving query response time due to our system is significantly greater than that achieved with an existing scheme such as page level caching using the same local space.
- Locally materializing data reduces the total work done in terms of time spent in transferring data between the mediator and user (response time) or remote sources and the mediator.
- The classes of materialized data evolve successfully with changes in the distribution of user queries to keep the mediator application optimized.

7.3 Applications for Experiments

We tested the effectiveness of the materialization system using three different Ariadne applications that we have developed. These applications are in different domains and integrate information from a variety of different Web sources. Testing the materialization system in different applications enabled us to demonstrate the effectiveness of the various aspects of the materialization system and validate the several hypotheses mentioned above. We present below an overview of the three Ariadne applications.

7.3.1 Information about Countries

As described in the Introduction, the information about countries mediator is an Ariadne application integrating information from the following Web sources:

- http://www.odci.gov/cia/publications/factbook/country.html (The CIA World Factbook) Provides interesting information about the geography, people, government, economy etc. of each country in the world.
- http://www.nato.int/family/countries.htm (The NATO Homepage) From which we can get a list of NATO member countries.
- http://www.un.org/Pubs/CyberSchoolBus/infonation/e_infonation.htm (InfoNation) Source which provides statistical data about UN member countries.

7.3.2 TheaterLoc

In the Introduction chapter we also mentioned TheaterLoc [Barish *et al.*, 1999] which is another Ariadne mediator application. TheaterLoc provides integrated access to Web sources about movies and theatres, an interactive map server depicting their various locations and a video server from which users can see video trailers of movies playing at the selected theatres. This application is available online at http://www.isi.edu/ariadne

Integrated access is provided to the following Web sources:

- Cuisinet. (http://www.cuisinenet.com) Web source providing information about restaurants in various US cities.
- Yahoo Movies. (http://movies.yahoo.com/movies/) Provides theater and movie showtime information.
- Hollywood.com. (http://www.hollywood.com) Movie previews source.
- E-TAK Geocoder. (http://www.geocode.com) Geocodes street addresses.
- US Census Map Server. (http://tiger.census.gov/cgi-bin/mapbrowse-tbl) Online interactive map server.

7.3.3 Flight Delay Predictor

Finally we have the Flight Delay Predictor application for performing flight delay predictions given information about a particular flight's departure and arrival times and airports,

airline name, weather predictions from the Yahoo weather service (http://weather.yahoo.com) and historical flight and weather data. A demo of this application is available at http://www.isi.edu/ariadne/demo/tw/ .

7.4 Experimental Results

We now present experimental results to validate the hypotheses outlined earlier.

7.4.1 Effectiveness of Source Structure Analysis

We tested the effectiveness of source structure analysis in all the three applications. We used the materialization system to identify and materialize data locally in these applications. We measured the improvement in query response time due to prefetching and materializing the data.

7.4.1.1 Countries Application

In the information about countries mediator the GUI is such that we can request one or more attributes of a country (such as area, coastline, population etc.) and also specify selection conditions on the 'name', 'region' and 'organization' attributes. The GUI specification for the countries application is shown in Figure 7.4.1.1.

```
SELECT {area coastline ...}
FROM countries
WHERE {name,1,(Albania,..)} {region,1,(Asia,....)} {organization,1,
(NATO,EEC,..)}
```

Figure 7.1: Specification for Countries application GUI

Given this GUI specification and the query processing axioms for the countries application the materialization system prefetches and materializes the following class of data: COUNTRY,(name,region,organization).
This is because of the fact that with the above interface the user can specify selection conditions on the non key attributes 'region' and 'organization'. Also the particular source that provides this data, the CIA World Factbook, is structured such that these selection queries are expensive. Given these facts the system decides to prefetch and materialize the primary key (name) and the selection attributes so that the selection can be performed

locally. This speeds up the processing of selection queries considerably (demonstrated below).

We used two query sets, Q1 and Q2 for this particular experiment. Q1 is a set of (200) queries that we generated and Q2 is a set of actual user queries to the countries mediator that was made available online for several months and for which we logged the user queries. In Q1 we introduced some distinct patterns in the queries. We measured the total query response time for the query sets Q1 and Q2 both without any data materialized and with locally materializing the above class of data after source structure analysis. The results are shown in Table 7.1 . As can be seen a very significant improvement in performance is achieved for both the query sets by locally materializing data based on source structure analysis.

Query set	Response Time (No optimization)	Response Time (With Materialization)	%improvement
Q1	38115 sec	9301 sec	75%
Q2	44186 sec	3775 sec	91%

Table 7.1: Source Structure Analysis for Countries Mediator

7.4.1.2 TheaterLoc

We ran several experiments to test the effectiveness of source structure analysis for materializing data in TheaterLoc. We ran experiments by considering different kinds of GUIs for the TheaterLoc application which had differences in the kind of data items that could be retrieved and also the kinds of constraints that could be specified.

We first assumed a GUI where the user would select from a list of cities and the TheaterLoc application would display the restaurants and theatres in that city on a map of the area. Given the GUI specification and axioms for the application, the materialization system prefetches and materializes the following:
RESTAURANT (place_name, latitude, longitude) and
THEATRE (place_name, latitude, longitude)

This is because in this particular application, for any query the mediator must plot the restaurants and theaters on a map which requires having the latitudes and longitudes of these locations. The latitudes and longitudes are obtained from an online geocoder (converts street addresses to latitude and longitude) which is structured such that only one

address may be geocoded at a time. As a result geocoding a set of places is very expensive and the materialization system decides to prefetch and materialize the latitudes and longitudes of all restaurants and theaters locally.

We present query response times (shown in Table 7.2) both without and with materializing data against a query set Q1 of queries that we generated for TheaterLoc.

Query set	Response Time (No optimization)	Response Time (With Materialization)	%improvement
Q1	22013 sec	9432 sec	57%

Table 7.2: Source Structure Analysis for TheaterLoc

As we can see, prefetching and materializing data due to source structure analysis greatly improves the performance of this application.

We then assumed a different GUI for the same TheaterLoc application. First we choose a bigger geographic area, i.e., say the entire Los Angeles area instead of choosing a smaller city in the area. Next we allow constraints on the 'price' attribute. So for instance we could now specify a query such as "Find all the restaurants in the Los Angeles area in the price range of $25 and below." As selection queries on the price attribute are expensive (the source for restaurants is structured such that we have to scan the pages of all restaurants to determine which ones are in the specified price range) the materialization system decides to prefetch and materialize the following class of data:
RESTAURANT (place_name, price)
This is addition to the classes identified earlier i.e.,
RESTAURANT (place_name, latitude, longitude) and
THEATRE (place_name, latitude, longitude)

We present query response times both without and with materializing the above classes of data against another query set Q1 of (30) queries that we generated for TheaterLoc. The results are shown in Table 7.3 . Clearly the materialized classes are very effective in improving the performance of the TheaterLoc application.

Query set	Response Time (No optimization)	Response Time (With Materialization)	%improvement
Q1	89093 sec	11550 sec	87%

Table 7.3: Source Structure Analysis for TheaterLoc

7.4.1.3 Flight Delay Predictor

For the flight delay predictor the system decides not to prefetch any class of data at all. The only kind of query that is ever asked in this application is to retrieve the delay prediction for a particular flight (by providing flight details such as the departure and arrival airports, airline, flight times etc.) The only Web source that the Flight Delay Predictor needs to access to answer the query is the Yahoo! weather source. However this is a source that may change any time and we need the most recent data from the source. Also the time taken to retrieve the weather prediction is not very high as only two pages (weather predictions for the departure and arrival airports) need to be retrieved from the remote Web source. Thus no data is prefetched.

7.4.2 Effectiveness of Extracting Patterns

We then tested the effectiveness of materializing the frequently accessed classes of data extracted as patterns by query distribution analysis. We measured the performance improvement due to materializing the frequently accessed classes of data (in addition to materializing data proposed by source structure analysis) in each of the three applications.

7.4.2.1 Information about Countries

For the countries mediator we again used the query sets Q1 (of generated queries) and Q2 (of actual user queries). Table 7.4 shows the improvement in query response time. There is a significant overall improvement in performance. There is also an improvement in performance over the case when data is materialized only on the basis of source structure analysis (Table 7.1) , it is more significant in the case of Q1 as there are distinct patterns that we introduced in the queries.

Query set	Response Time (No optimization)	Response Time (With Materialization)	%improvement (total)	%improvement (source structure)	%improvement (query distr.)
Q1	38115 sec	1549 sec	96%	75%	21%
Q2	44186 sec	2174 sec	95%	91%	4%

Table 7.4: Effectiveness of Extracting Patterns in Countries Mediator

7.4.2.2 TheaterLoc

The results for the effectiveness of materializing frequently accessed classes of data for TheaterLoc are shown in Table 7.5. In this application as well there is a further improvement in performance over the case where classes are materialized just on the basis of source structure analysis (Table 7.2).

Query set	Response Time (No optimization)	Response Time (With Materialization)	%improvement (total)	%improvement (source structure)	%improvement (query distr.)
Q1	22013 sec	1644 sec	93%	57%	36%

Table 7.5: Effectiveness of Extracting Patterns in TheaterLoc

7.4.2.3 Flight Delay Predictor

For the flight delay predictor we did not materialize any data at all based on source structure analysis. 7.4.2.3 shows the improvement in performance due to materializing frequently accessed classes. The performance gain of 34% (over no materialization) is substantial. However it is significantly less than the performance improvement achieved for the countries and TheaterLoc applications. This is because all queries for the Flight Delay Predictor require fetching just 2 pages from one Web source (retrieving the weather predictions from the Yahoo! weather service). As a result the queries to the Delay Predictor even without any materialization do not take a very long time to execute.

Query set	Response Time (No optimization)	Response Time (With Materialization)	%improvement (total)	%improvement (source structure)	%improvement (query distr.)
Q1	2399 sec	1603 sec	34%	0%	34%

Table 7.6: Effectiveness of Extracting Patterns in Flight Delay Predictor

7.4.3 Comparison with Page Level Caching

The third hypothesis states that the performance improvement achieved by our system will be greater than an existing caching scheme such as page level caching. We implemented a page level caching system for Ariadne and compared the performance improvement achieved by our system with that achieved by page level caching using the same local space.

The results are presented in Table 7.7 for the countries application, in Table 7.8 for TheaterLoc and in Table 7.9 for the Flight Delay Predictor.

Query set	Response Time (No optimization)	Response Time (Our System)	Response Time (Page Level)	%improvement (Our System)	%improvement (Page Level)
Q1	38115 sec	1549 sec	34320 sec	96%	10%
Q2	44186 sec	2174 sec	37993 sec	95%	14%

Table 7.7: Comparison with Page Level Caching for Countries Mediator

Query set	Response Time (No optimization)	Response Time (Our System)	Response Time (Page Level)	%improvement (Our System)	%improvement (Page Level)
Q1	22013 sec	1644 sec	17832 sec	93%	19%

Table 7.8: Comparison with Page Level Caching for TheaterLoc

Query set	Response Time (No optimization)	Response Time (Our System)	Response Time (Page Level)	%improvement (Our System)	%improvement (Page Level)
Q1	2399 sec	1603 sec	1846 sec	34%	23%

Table 7.9: Comparison with Page Level Caching for the Flight Delay Predictor

For the countries mediator, the performance optimization achieved by our materialization system is much greater than with page level caching. This is because for this particular application most of the data is retrieved from the CIA World Factbook source in which each page is fairly large containing information on various attributes of a country. With page level caching we are committed to storing all (or none) of each page and as a result the local space for materialized data is not used optimally. With our system we have much more flexibility in selecting the information on each page that must be materialized and thus the local space is used much more efficiently. For TheaterLoc also the performance improvement due to our system is substantially greater than with page level caching. In the case of the Flight Delay Predictor though the improvement due to our system over improvement due to page level caching is not as high as the other applications. This is because in this application the only pages we retrieve are those of weather predications from Yahoo!. From these pages we extract all the information about the weather prediction (i.e., forecast for that day and following few days). Now storing the entire Web page takes only a little more space than storing the data that we *extract* from the page.

In our system we end up storing the extracted data, in page level caching we store the entire pages (queried frequently). Thus there is not a very high difference in performance improvement due to our system vs page level caching.

7.4.4 Updates

The fourth hypothesis states that the total work done (in terms of time spent in transferring data) is less in the case when we locally materialize data. We set up an experiment (for the countries mediator application) where we vary the update frequency (artificially) of the CIA World Factbook source from 0 to 20 updates per day. For each frequency we measure the total work done with and without materialization for query set Q1. The total work done with materialization is the sum of the user query response time and the time spent to keep the materialized data up to date. The total work done without materialization is simply the user query response time.

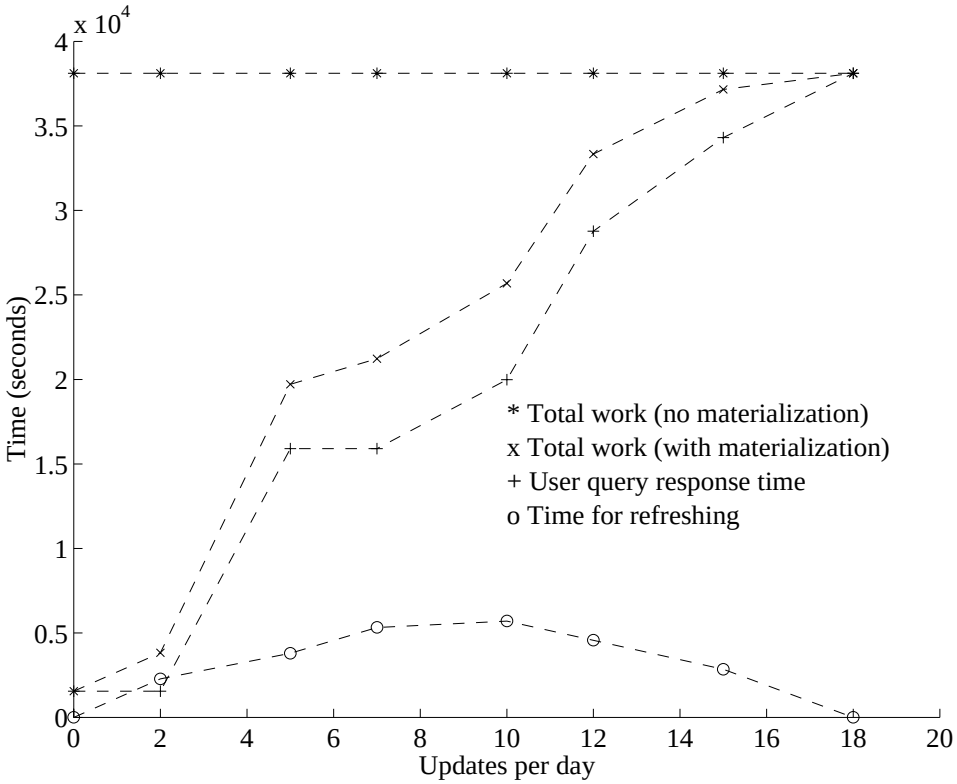


Figure 7.2: Work done with and without materialization

The results are shown in 7.2. As we can see, for all update frequencies the total work done with materialization is less than the total work done without materialization. As the

update frequency becomes higher, the system materializes fewer classes of materialized data as the total maintenance cost gets higher.

7.4.5 Evolving Query Distribution

The final hypothesis is that the materialization system adapts to changes in the user query distribution. Specifically if the patterns of frequently accessed classes of data change then the materialization system locally materializes the new more frequently queried classes of data, replacing existing materialized classes that are not queried frequently.

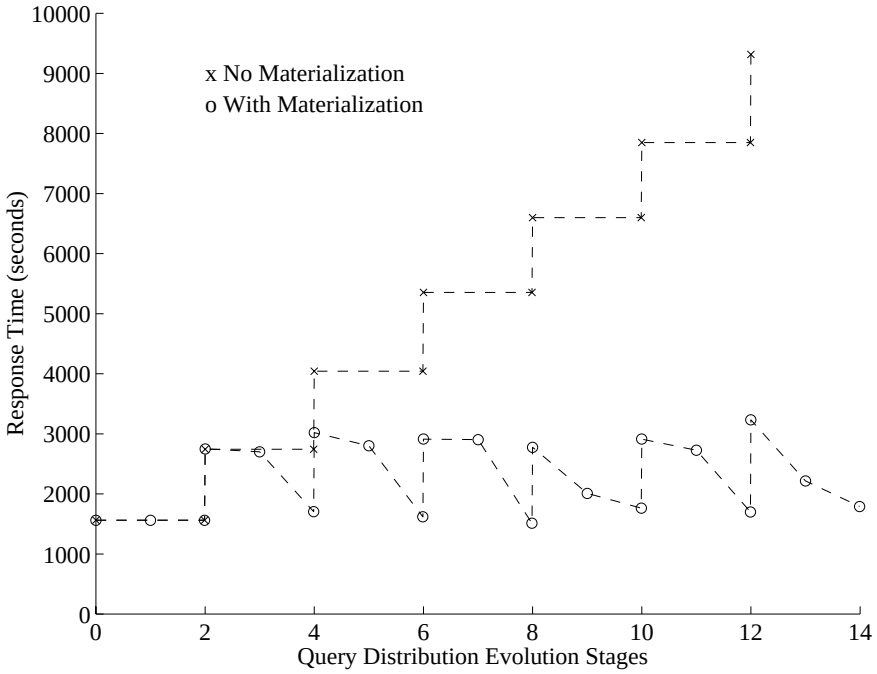


Figure 7.3: Evolving with the Query Distribution

We set up an experiment for the countries mediator where we started with an existing query distribution Q1 of 200 queries. Q1 is such that it has a pattern of 6 frequently accessed classes of data. We then incrementally changed the query distribution to one having a pattern of 6 entirely different classes of frequently accessed data. We would at each stage of change remove an existing class in the pattern and insert another different class of frequently accessed data in the pattern. For instance starting with a distribution having a pattern {C1,C2,C3,C4,C5,C6} representing classes of frequently accessed data, we would change the pattern to {C1',C2,C3,C4,C5,C6}, then to {C1',C2',C3,C4,C5,C6} etc.

The materialization system adapts to changing query patterns by periodically re-evaluating the query distribution, inserting new more frequently accessed classes of data into the materialized data store and replacing existing classes if necessary. To evaluate whether the materialized data store indeed successfully evolves with a changing query distribution, we changed the query distribution incrementally as described above. We then measured the total query response time for the query set Q1 (as it evolves) both with and without the materialization system doing re-evaluation of the query distribution (and adding and replacing classes of materialized data if necessary). The results are shown in 7.3. The response time is measured at the stages when the query distribution is incrementally changed. These are the even numbered stages 2,4,6, etc. It is also measured between the stages when the query distribution is changed i.e., at the stages 1,3,5 etc.

We begin at the stage when the classes of data in the initial pattern in Q1 are materialized and the response time is quite low. Let us look at the response times without re-evaluation. The total query response time suddenly increases at each stage when we change the query distribution. This is because there is now a new class of frequently accessed data in the distribution but is not materialized locally. As a result the response time increases each time the distribution changes. As the system does not adapt to changes with changes in the query distribution, performance just degenerates with a changing distribution. In the case where the materialization system does re-evaluate the query distribution (and makes changes to the materialized data store if necessary) the response time does not increase monotonically as in the previous case. While the response time does increase when the distribution changes, re-evaluation causes the new class of frequently accessed data to be eventually materialized locally thus reducing the response time. This can be seen graphically in 7.3 where though the response time increases each time the distribution is changed, it is eventually lowered as the materialized data store evolves. This experiment thus validates our hypotheses that the materialization system is indeed able to adapt to a changing query distribution.

7.5 Summary

We presented experimental results demonstrating the effectiveness of a variety of aspects of our materialization system. We tested the system using three different Ariadne applications. The effectiveness of different portions of the materialization system can vary with the application. As a whole however the materialization system is indeed very effective in optimizing performance for any application. The total work done in terms of data transfer

is less when data is locally materialized. Finally the materialization system is able to successfully adapt to changes in the user query distribution.

Chapter 8

Related Work

In this chapter I first discuss how my work relates to several other research efforts that also aim at optimizing performance of various systems such as databases, operating systems or Web servers by materializing or caching data. I then describe in detail how my approach can be applied to several other mediator systems that are based on an architecture similar to that of Ariadne.

8.1 Related Work

Improving performance by materializing or caching data is a topic that has been extensively investigated in the context of client–server database systems, operating systems and more recently for proxy servers for the Web. Also although there are several efforts in progress on building information integration mediators, very few have addressed the issue of improving performance by materialization or other means. The problem is also similar to the view selection problem when deciding what views to materialize in a data warehouse. I now provide a comparison of my work with these efforts.

8.1.1 Semantic Caching in Databases

Most work on client-server database-systems caching and operating-systems caching has focused on tuple-level or page-level caching. Only recent approaches are based on semantic level [Dar *et al.*, 1996] or *predicate based* [Keller and Basu, 1996] caching. Semantic or predicate-based caching is a form of caching where portions of data that can be described using predicates are cached. For instance consider an EMPLOYEE relation in a database with attributes name, age, salary, address. We could cache a portion of this relation such as *" the names and addresses of all EMPLOYEEs earning more than $50,000 per year"* .

This can be described using the following predicate based description:
$EMPLOYEE(name, \_, \_, address)$ AND $salary > 50000$

The advantage of semantic caching (over tuple-based or page-based caching) is that we are more flexible in terms of specifying exactly what to cache. A semantic caching system also includes a mechanism for *containment checking* i.e., given a user query, it uses the predicate descriptions of the cached data to determine what portion of the query can be answered using the cached data. As far as representation and use of materialized data are concerned, our approach is quite similar to that of a semantic caching system. Using LOOM we provide a semantic description of the cached data. Also the query planner reasons with these descriptions when generating a plan to answer a user query. As the planner also determines what portions of the query may be answered using the materialized data sources it does the containment checking step in the process of query planning.

A problem with the semantic caching approach is that that the containment checking problem i.e., determining exactly what portion of the data requested in a user query is present in the cache, is hard and having a large number of semantic regions creates performance problems. A solution proposed in [Keller and Basu, 1996] is to reduce the number of semantic regions by merging them whenever possible. This is in fact an idea we have built on. In the CM algorithm we have presented an approach for systematically creating new semantic regions to consider for materializing and merging them when possible.

8.1.2 Caching in Web Servers

Caching schemes have also been proposed for Web proxy servers [Chankunthod *et al.*, 1995] to reduce the amount of data that must be shipped over the internet. However these approaches are based on caching at the level of individual Web pages, which we argued is not optimal in our environment. In fact we have demonstrated through experiments that our approach does indeed out perform caching or materializing at the level of entire Web pages.

8.1.3 Caching in Federated Databases

An approach to caching for federated databases is described in [Goni *et al.*, 1997]. Theirs is also a semantic caching approach where the data cached is described by queries. They also define some criteria for choosing an optimal set of queries to cache. Finding the optimal set is an NP-complete problem and they use an A* algorithm to obtain a near optimal solution. A limitation of their approach is that the cached classes can only be in terms of

classes in a predefined hierarchy of classes of information for a particular application. My approach is much more flexible in that we dynamically construct classes of information to materialize.

8.1.4 Materialized Views

My work is also related to recent work on view selection in a data warehousing environment. One of the most important decisions in designing a data warehouse is selecting what views to materialize so that the total query response time is minimized with a constraint such as limited storage space and/or cost of maintaining the views. [Yang *et al.*, 1997] and [Gupta and Mumick, 1998] show that this is an intractable problem and present heuristic algorithms for near optimal solutions. However the warehousing problem differs from my problem in the following aspects:

(i) There is a fixed set of views in the warehousing problem and a decision is to be made for each view whether to materialize or not. In the mediator environment we dynamically propose new views to materialize. Also the number of views proposed must be kept small from a query processing perspective.

(ii) In the warehouse environment the views represent either base tables at the individual databases, user queries to the warehouse or intermediate results of computing user queries from the base tables by applying relational algebra operations one at a time. In the mediator environment however the "views" are actually classes of data defined in terms of the mediator model as described earlier.

(iii) Query planning is not an issue in warehousing as the number of views is small and queries are to particular views. In the mediator problem however we must ensure that the query planning cost is kept small by keeping the number of new views created small.

There are also differences in the update aspect of the problem i.e., taking the maintenance cost into account when materializing views and strategies for keeping the materialized data consistent. They are summarized below as follows:

(i) The costs that are to be minimized. In the warehouse environment the primary update cost (that we would like to keep limited) is the cost of scanning large relations from the disk into main memory and main memory processing on large tables. As such most work on optimizing maintenance of materialized views in the warehouse environment has focused on incrementally maintaining materialized views [Gupta *et al.*, 1993] (as opposed to fully recomputing the views) in order to save processing time for view maintenance. There is further work on "self-maintainable" views [Quass *et al.*, 1996] where auxiliary data (typically keys of relations etc.) is also materialized in order to save the cost of

fetching that data from the disk when updating a materialized view. In the mediator environment however the primary cost is that of fetching data from remote sources such as fetching pages over the internet and the cost of main memory processing is relatively small. We must thus ensure that the data fetched from the remote sources for updates is kept limited. There being an order of magnitude difference between the cost of retrieving data from remote sources and the cost of main memory processing, it makes sense to first minimize the data retrieved from the remote sources and only then consider further optimizations for main memory processing.

(ii) Maintenance strategies in case of updates. In the warehouse environment typically changes to the relations in individual databases (deltas) are sent to the warehouse, the changes are smaller than the entire updated relation and thus cheaper to send. The algorithms developed for view maintenance can update the materialized views given only the changes and the existing views. Algorithms have also been developed for incremental maintenance of views over semi-structured data [Abiteboul *et al.*, 1998] and it is shown that incremental maintenance is cheaper than fully recomputing the views in most cases. In the mediator environment however determining the change to relations in a remote source such as a Web source is often as expensive as retrieving the entire new relation and the cost of recomputing the entire view will only be marginally more than that of incremental maintenance.

(iii) Detecting changes at the sources. In the warehousing environment the time and frequency of changes to the sources databases is given and triggers are sent from the source databases to the mediator notifying the changes. In the mediator environment we determine the frequency with which each materialized class needs to be updated given the update characteristics and user's freshness requirements, and simply recompute the materialized class with that frequency.

8.1.5 Mining Association Rules in Data Mining

Finally the problem of extracting patterns from queries is somewhat similar to the problem of data mining, particularly that of mining *association rules* [Agrawal *et al.*, 1993]. The problem of mining association rules is that of extracting implications of the form $X \Rightarrow Y$ from a database where $X \subset I$ and $Y \subset I$ and $X \cap Y = \phi$, where $I = \{i_1, i_2, ...i_m\}$ is a set of data items. The patterns that we extract from queries can also be looked upon as implications. For instance a pattern such as "economy and population of European Countries " is an implication European Countries $\Rightarrow$ (economy, population). Extracting patterns however differs from mining association rules in several respects. First, when

mining association rules, the antecedent X in an implication $X \Rightarrow Y$ could be any $X \subset I$ where at least a certain minimum percentage of transactions in the database contain $X \cup Y$. In extracting patterns the antecedents are just the individual classes and subclasses of data queried. Second, we do not cluster data items in the consequents in the implication rules whereas in extracting patterns we do try and group together data items in consequents (for the same antecedent) that overlap well. Third, when extracting patterns we can extract patterns when the antecedents (classes and subclasses) are organized in a class/subclass hierarchy. This hierarchy is dynamically generated. In mining association rules however the hierarchy in which antecedents may be organized is predefined and fixed [Srikant and Agrawal, 1995]. Finally mining association rules is often done for databases of very large size and the optimizations focus on issues like minimizing the disk scans of the relations for data mining. In extracting patterns the entire query distribution being analyzed can fit into main memory and the optimizations for minimizing disk scans are not required. For all the above reasons the algorithms for mining association rules cannot be directly applied to the problem of extracting patterns which is why we developed the CM algorithm for this task.

8.2 Applicability to Other Mediators

I now provide a discussion on how the ideas of my thesis can be applied to several other related mediator systems that have been built by other research groups. All of these systems perform essentially the same task as SIMS and Ariadne i.e., integrating data from databases or Web sources, also the high level mediator/wrapper architecture is similar across these systems. It is thus worth while considering how the performance optimization by materialization approach presented in this thesis would apply to these other systems.

The most important issues when considering applying our materialization system to other mediators are:

- Representation and query planning issues. How will the idea of defining the materialized class of data as another source apply in that system? What will be the impact on query processing, i.e., how do we ensure that materialized data is effectively used?
- How will the ideas for selecting data to materialize based on source structure analysis, query distribution analysis and updates apply?
- Can our approach to updates at Web sources be incorporated in that system?

We now discuss each of the above issues in detail.

8.2.1 Representation and Query Planning

For each of the other mediator systems I will first provide an overview of the information modeling and query reformulation aspects of the system as these are central to understanding how our materialization approach can be applied. We then discuss if and how we can represent the materialized data as another source in that system and also the impact on query planning.

8.2.1.1 Information Manifold

The Information Manifold [Levy *et al.*, 1995] is an implemented system that provides uniform access to structured information sources on the World Wide Web. A user is presented with a *world-view* which is a set of relations (including a class hierarchy) modeling the kinds of information that can be obtained from the available sources. For instance consider again the restaurant example that we have been using throughout. The world-view includes the relation Restaurant(name, address, city, telephone, cuisine, review, rating) providing an integrated world view of restaurants. The world view also contains a hierarchy of classes as shown in 8.1.

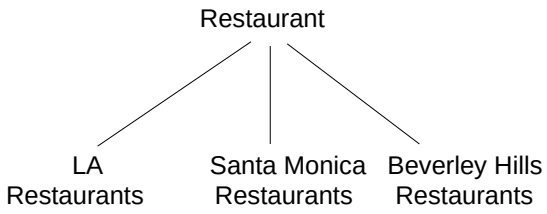


Figure 8.1: Hierarchy of classes in world view

The system also contains descriptions of the contents of the individual information sources which contain the actual data. The representation language used for the world view and information source descriptions is an extended description logic called CARIN-CLASSIC. The information sources are described in terms of the world view relations. Each source is described by a conjunctive formula which describes the constraints satisfied by facts in the relation found in the source. For instance suppose we have a source that provides information about "Santa Monica" restaurants. We can model it as:

```
v(name,address,city,tel,cuisine,review,rating): Restaurant(name,address,city,tel,cuisine,
review,rating) ^ city="Santa Monica"
```

We could describe a materialized source of data in a similar fashion. For instance suppose we materialized the name and address of all chinese restaurants. We describe this materialized source as:

```
v(name,address): Restaurant(name,address,city,telephone,cuisine,review,rating)
^ cuisine="Chinese"
```

We now need to examine how the query processor will use the contents of the materialized data sources. In Information Manifold the queries to the system are formulated in terms of world view relations, however the system must generate a *query plan* that uses the information sources to answer the user query. The query reformulation algorithm first decides which sources are *relevant* given a user query. This is done by reasoning with constraints provided in the query as well as the information source description on the basis of which some sources may be pruned as not relevant to answering the query. Next the algorithm computes *conjunctive plans* by considering the possible combinations of relevant sources. The algorithm is designed to generate **sound plans** i.e., all the answers it produces are guaranteed to be answers to the query. The algorithm also generates **minimal** query plans, a query plan is minimal if we cannot remove a subgoal from the plan and still obtain a sound plan. The algorithm generates minimal query plans in order to answer the query with accessing the minimum possible sources. Now the query processing algorithm will also consider a materialized data source in generating a query plan. However a minimal query plan may be a misleading criterion for estimating the cost of a query plan. What we would need the algorithm to do is to consider *all* the possible conjunctive query plans, compare the costs of the query plans using a cost estimator and choose the plan with minimum cost. The question to ask then is whether the Information Manifold algorithm query reformulation algorithm produces all the necessary conjunctive query plans. This is closely related to the problem of answering queries using materialized views. The problem of answering queries using views has however been shown to be NP-complete in the number of information sources. So as in the case of SIMS, creating too many information sources for materialized data will cause performance problems for the query planner in Information Manifold too. Our solution to the problem by creating just a few new information sources by describing the data more compactly is applicable here also. By describing the materialized data classes compactly we create few new information sources in the system. To summarize the above, the materialized data can be described as an auxiliary information source in the Information Manifold system too as shown above. The query processor however needs to be extended to take cost information into account

also so as to prefer to use the materialized data sources. Also creating too many new information sources will cause performance problems for the query planner and a compact description will help create just a few sources.

8.2.1.2 Infomaster

Infomaster [Genesereth *et al.*, 1997] is another information integration system. The representation of the integrated world view and the information sources is done in a manner very similar to that in Information Manifold. We have *world relations* that are global relation names against which the user poses queries. For instance for the restaurants domain we could have the world relation:

```
Restaurant(name, address, city, telephone, cuisine, review, rating)
```

Source are described by *source relations*. Relationships between source and world relations are specified by describing each source relation as the result of a conjunctive query over the world relations. For instance for the restaurant domain, a source providing information about restaurants in Santa Monica would be described as the view:

```
CREATE VIEW Santa_Monica_Restaurants
SELECT name, address, city, telephone, cuisine, rating, review
FROM Restaurant
WHERE city = ``Santa Monica''
```

We can now describe a class of materialized data, say the name and address of all chinese restaurants as an auxiliary information source as the view:

```
CREATE VIEW Chinese_Restaurants
SELECT name, address
FROM Restaurant
WHERE cuisine = ``chinese''
```

In this framework thus the problem of finding a query plan given a user query, is the same as the problem of answering queries using views. A distinguishing feature of the query planner in Infomaster is that they also find *maximally contained query plans*. It may be the case that not all the information requested by a user can be answered given the set of information sources available, however a *portion* of it may be answered. As opposed to other systems where we find a query plan which is exactly equivalent to the given user query, Infomaster finds maximally contained query plans i.e., plans that

return as much of the user's query as can be answered given the available sources. The system exploits *functional dependencies* between attributes of various relations and is able to generate *recursive* query plans. It is shown that recursive query plans are necessary in order to have maximally contained plans. The system is also able to generate query plans with disjunction. Infomaster also considers the problem of query plan optimization. An information source $s_i$is represented by two views: v_i which is a lower bound of s_i and v_i which is an upper bound of s_i. The query processing algorithm of Infomaster generates what they call *semantically correct*, *source complete* and *view minimal* query plans. A query plan P is semantically correct with respect to a user query Q if P is contained in Q for all instances of the source relations $s_1, s_2, ..., s_n$ consistent with the given conservative and liberal views. A query plan P is said to be source complete if every semantically correct query plan P' is contained in P for all instances of the source relations $s_1, s_2, ..., s_n$ consistent with the given conservative and liberal views. For optimizing the query the system must consider the cost of coming up with the answer. The Infomaster system uses the notion of *view minimality*. A query plan P is view minimal if every semantically correct and source complete plan P' queries at least as many information sources as P. Infomaster generates semantically correct, source complete and view minimal query plans. Briefly the query planning algorithm works as follows. In the first step a semantically correct and source complete query plan is generated using one of the algorithms in [Levy *et al.*, 1995] . In the second step, redundant parts of this query are eliminated to produce view minimal plans. As in the case of Information Manifold it can be argued that view minimal query plans may not necessary be the cheapest and the system must compare the *costs* of executing various semantically correct and source complete plans and choose the one which is least expensive to execute. Creating too many sources for materialized data will cause performance problems for the query planner in this system as well.

8.2.1.3 TSIMMIS

TSIMMIS [Hammer *et al.*, 1995] represents a *query centric* approach to information integration which is different from the approach of the above Information Manifold or Infomaster systems. In TSIMMIS we specify using a mediator specific language how a query can be answered using the various information sources available. Assume that we have two sources providing information about restaurants, namely zagats and fodors. The sources can be accessed through wrappers around the sources (say we name the wrappers zagatswrap and fodorswrap respectively). Both the sources *export* i.e., provide information about restaurants. Say the zagats source exports the relation zagat_restaurants and the

fodors source exports the relation fodors_restaurants. We can provide an integrated view of a relation restaurants by specifying:

```
<restaurant {<name N> <address A> <review R>}> :-
<zagats {<restname N> <address A> < review R>}>@zagatswrap OR
<fodors {<name N> <location A> <review R>}>@fodorswrap
```

In the TSIMMIS approach we thus directly specify how various information sources can be combined to answer a user query. The advantage is that no query planning is really required as the plan of how to combine sources is already laid out in the mediator specifications. The disadvantage is that the approach is not at all suited for cases where information sources may be added or deleted frequently as the entire mediator specification must be rewritten with each change. Suppose we did materialize the names and reviews of all chinese restaurants we could include the materialized data source by specifying:

```
<chinese_restaurant {<name N> <review R>}> :-
<chinese_restaurants_materialized {<name N> <review R>}>@chineserestwrap
```

We have to create one such specification for each materialized data class. A major limitation in TSIMMIS is that there is no facility for query containment checking. Unlike SIMS, Information Manifold or Infomaster where the query planner can determine what portion of a user's query can be retrieved from the materialized data sources, in TSIMMIS user queries are posed directly to classes that the mediator makes available. As the user is limited to posing queries to only these classes query containment is not required to be done at all. Given this, our approach would not apply well to the TSIMMIS system which is more suited for cases where the user has a good idea in advance of what classes are to be queried and the set of information sources does not change frequently.

8.2.1.4 InfoSleuth

InfoSleuth [Bayardo *et al.*, 1996] is a system for information integration from multiple sources based on an agent architecture. We provide a brief description of the primary agents in the system. The User Agent helps in formulating user queries and displaying results. The Ontology Agent answers queries about ontologies. The Broker Agent records advertisements from all InfoSleuth agents on their capabilities and responds to queries from

agents as to where to route their specific queries. Associated with each information source is a Resource Agent that provides a mapping from the common ontology to the database schema and language native to that source. The Task Execution Agent coordinates the execution of high level information gathering subtasks. Query planning is treated as a planning problem in InfoSleuth in much the same way as in the SIMS system. The Task Execution Agent uses a query planner for generating information gathering plans that is very similar to the query planner of SIMS.

Our materialization approach would apply well to the InfoSleuth system. For each class of materialized data we simply need to create a new Resource Agent and register its capabilities with the Broker Agent. Now the materialized data source is also considered when the query planner is generating an information gathering plan. Our approach of having compact descriptions for materialized data and thus creating fewer new sources will be helpful as a large number of new sources will create performance problems in InfoSleuth as well.

8.2.1.5 DISCO

DISCO [Tomasic *et al.*, 1997] is another mediator system for integrated access to multiple heterogeneous database systems. To describe the integrated world view DISCO uses a data model based on the ODMG-93 data model specification. For instance an integrated view of restaurants could be defined as the class:

```
interface Restaurant {
attribute string name;
attribute string address;
attribute string cuisine;
attribute string review; }
```

The contents of an information source are specified by specifying the relation it provides along with the name of the wrapper for accessing its data and the name of the data source. Also we specify a mapping between the source relation and attributes and a class and attributes provided by the mediator.

For instance suppose zagats was one source providing information about the Restaurant class. We would specify this as:

```
extent restaurant0 of Restaurant wrapper w0 repository r0
map ((zagats0=restaurant0), (name=n), (address=a), (cuisine=c), (review=r));
```

In this framework we can define a class of materialized data too as another source in a similar fashion. For instance a materialized data for the name and address of chinese can be incorporated by first defining a subclass of Restaurants as:

```
interface Chinese_Restaurant: { Restaurant(cuisine= ``chinese'') }
```

and the source definition as:

```
extent chineserestaurant0 of Chinese_Restaurant wrapper w1 repository r1
map ((matchinese0=chineserestaurant0), (name=n), (address=a));
```

The DISCO query processor uses a query reformulation process on the lines of [Kirk *et al.*, 1995] in query planning and optimization. So in this system also it is advisable not to create too many such new information sources for materialized data and instead used a compact description creating few new sources.

8.2.1.6 Garlic

We finally mention Garlic [Haas *et al.*, 1997] , which is a system and set of tools for the management of large quantities of heterogeneous multimedia information. An integrated view over several heterogeneous and multimedia sources is provided through a unified schema expressed in an object-oriented data model. It can be queried and manipulated using an object-oriented dialect of SQL. The data in individual data sources is made available by wrappers around these sources. Each wrapper turns the data available in its underlying repository into objects accessible by Garlic. Each Garlic object has an *interface* that abstractly describes the object's behaviour. Wrappers provide a description of their contents using the Garlic Data Language (GDL) which is a variant of ODMG's Object Description Language (ODL). For instance a source providing information about Restaurant objects (say an information source such as the zagats Web source) would describe its contents as:

```
interface Restaurant {
attribute string name;
attribute string address;
attribute string cuisine;
attribute string review;}
```

The Garlic framework is such that a new information source can be incorporated with ease and without changing existing portions of the description as has to be done in TSIMMIS. A materialized data source (continuing with our example of the name and address of chinese restaurants) can be described similarly. We create a subclass of Restaurants (say Chinese_Restaurants) in the object-oriented model for the restaurant domain. The materialized data source can then be described as:

```
interface Chinese_Restaurant{
attribute string name;
attribute string address;
}
```

The issue of the impact on query processing now remains. Garlic has a query planning process quite different from any of the above mediator systems. Query planning involves the stages of parsing, semantic checking, query rewriting and query optimization. Garlic is one of the few systems that takes cost-estimates into account in generating query plans. STARs(STrategy Alternative Rules) are used in the query optimizer to describe possible execution plans for a query. The optimizer uses dynamic programming to build the query plans bottom up. First, single collection access plans are generated followed by a phase in which 2-way joins are considered, followed by 3-way joins etc., until a complete query plan for the query has been chosen. Although this dynamic programming method scales well to a large number of information sources the process is still exponential in the information sources. As with other mediator systems again it is dis-advantageous to create too many new information sources and a compact description will be more helpful.

To summarize, materialized data can indeed be represented in most of the above systems (with the exception of TSIMMIS) as another information source. Creating a large number of sources causes performance problems in all such systems. This is expected as the general problem of query planning to gather information from multiple sources is combinatorially hard and performance will remain an issue no matter what the source representation and query language. We also note that not all of these systems have cost-based query optimizers that can select a low cost query plan from several alternative query plans. The query planner for any of the above systems will need to be augmented to perform cost based query optimization in case it does not do so, to ensure that it generates plans using the materialized data.

8.2.2 Selecting Data to Materialize

The ideas for source structure analysis and query distribution analysis are general and can be incorporated in any of the above systems to select classes of data to materialize. Not all the above systems use query processing axioms as in SIMS that are used in source structure analysis. Alternative approaches to incorporating source structure analysis would involve either extending these systems to use precompiled axioms or develop modified source structure analysis procedures that do not use axioms.

8.2.3 Updates

The ideas for keeping the materialized classes of data up to date by periodically refreshing them and also the update specifications can be incorporated into any of these systems. The update approach is independent of the modeling and query planning approaches of a mediator system and thus can be easily applied in any mediator system.

8.3 Summary

We discussed how the work in this thesis compares with other efforts on optimizing performance by similarly materializing or caching data. Materialization or caching in several different environments such as databases, Web servers and federated databases was considered. We then described in detail how our approach can be applied to several other mediator systems that are based on an architecture similar to that of Ariadne.

Chapter 9

Conclusion

I provide a conclusion of my thesis work followed by directions for future work.

9.1 Conclusion

The high query response time is an issue in information mediators. The response time is high despite having efficient query planners in mediators as often a large number of Web pages have to be fetched from remote sources to answer a user's query. This dissertation presents an approach to addressing the performance issue by locally materializing data. We have presented a framework for materializing data in an information mediator environment, argued that data must be selectively materialized and presented an approach for automatically selecting data to materialize. We have implemented a materialization system for the Ariadne mediator based on the ideas in this dissertation and demonstrated its effectiveness in optimizing performance in several applications. The general approach is applicable to other mediator systems that have been built using a similar architecture. Experimental results demonstrate that the materialization system built on the ideas in this thesis is indeed effective in optimizing performance. I now discuss some directions for future work in this area.

9.2 Future Directions

The directions for future work from this research fall in two main categories. First are directions for further improving or enhancing the existing functionalities of the materialization system. They include adding support for handling multimedia types of data, investigating interaction between the query planner and materialization system and automating the collection of data for cost estimation. There are aspects of my thesis work

that I see benefiting areas in databases and information management in addition to optimizing mediator performance. For instance some of the ideas can be applied to semantic caching, E-commerce, and semi-structured data management. I will now elaborate on each of these.

9.2.1 Support for Multimedia Data

One direction in which mediator technology is advancing is to enable information mediators to extract and integrate data from multimedia types of sources such as multimedia data (text and images) from Web sources and data from text databases, image or video databases that can be queried by content. Along with other components of a mediator system there is then a demand for extending the materialization system as well to handle multimedia types of data. An issue is that data like text, images or a video sequence often takes up a lot more (local) space as compared to structured data. One solution could be to materialize a summary of text for textual data or image thumbnails instead of the full image so as to materialize a "summary" of the data without consuming too much space. I plan to investigate other aspects as well in order to extend the materialization system for multimedia data.

9.2.2 Interaction with Query Planner

The materialization system relies on the query planner to make the maximal use of the materialized data in order to generate high quality plans. Our experiences show that that a query planner capable of cost based query optimization does indeed generate plans that use the materialized data in most cases. However there is no guarantee that this is always the case i.e., the materialized data will be used optimally. A tighter coupling between the query planner and the materialization system can be a solution to this problem. The approach will be easier to develop in systems like Ariadne where precompiled axioms exist that specify how data for a domain class can be obtained from one or more sources. As a simple example, consider the axiom:

```
country(name,region,gdp,population,....) <->
ciafactbook(name,region,gdp,population,...)
```

Now suppose we materialized the primary key(name) and the attribute region of the class country so that selection on the country attribute can be done locally. Instead of relying on the query planner to generate plans that will use the materialized data to do selection on the region attribute locally, we could simply rewrite the above axioms as:

```
country(name,region,gdp,population,....) <-> region_materialized(name,region) and
ciafactbook(name,gdp,population,...)
```

This now *ensures* that the query planner will indeed use the materialized data to perform selection on region locally. Another area of future work is thus to develop a tighter coupling between the query planner and materialization system for Ariadne as well as other mediator systems.

9.2.3 Automatically Collecting Specifications

At present we assume that we know apriori the time and frequency with which a particular Web source changes. However for many sources, such as say a Web source providing information about the computer science faculty at a university, the source may change at any time. Techniques have been developed for automatic change detection [Chawathe *et al.*, 1996] in textual data, with a focus on semi-structured data from Web sources. It would be useful for the materialization system to use such systems for detecting changes at the Web sources instead of simply refreshing the data with an appropriate frequency for that source.

We manually specify statistics about Web sources that enable the cost estimator to estimate costs of various queries to the Web sources. There is recent work on developing more sophisticated cost models for wrapped Web sources and also learning response times [Bright *et al.*, 1999] for Web sources. We could use these systems to automatically collect data (such as the time to retrieve a page from a particular Web source) that will enable us to automatically assemble the statistics needed for a cost estimator instead of having to specify this information manually.

Currently we also manually specify the space occupied by various attributes and the size (number of tuples or objects) of a particular class. The collection of such data can be automated by having a program to estimate class size by say queries that request for all tuples in a class and attribute size by averaging over the size of a particular attribute in different tuples or objects.

9.2.4 Semantic Caching

As described earlier our materialization approach is a kind of semantic caching in the mediator environment. The problem of containment checking is a hard problem is semantic caching systems in databases. Our approach of systematically merging materialized classes to form more compact and fewer classes can be a solution to this problem. I plan to

investigate how classes that we consider for caching in database systems can be merged and made more compact in a fashion similar to merging classes in CM. I will also evaluate how this approach can address the containment checking problem for semantic caching and compare it with other approaches for addressing the problem such as using approximate cache descriptions or indexing the cached predicates.

9.2.5 Query Mining

The extraction of patterns in queries by CM can be viewed as "query mining" on a query distribution. Extracting such patterns from user queries could be of interest in areas besides materialization. One example is E-commerce where sellers may be interested in analyzing online user queries and request to find classes of items (data) that users are most interested in. The user profile may also be analyzed in addition to the queries. An example of a useful pattern extracted for an online book store may be "college student customers interested in humor/fiction mostly buy books by PG Wodehouse or Evelyn Waugh". It would thus be useful to investigate how an algorithm like CM may be enhanced or modified to satisfy the needs of query mining in applications like E-commerce. The main difference from data mining is that data mining is done on data whereas query mining is performed on queries. Some enhancements needed for CM for query mining could be learning more sophisticated descriptions and also augmenting the analysis of the user query distribution with other inputs such as customer profiles.

9.2.6 Querying Semi-structured Data

The interest in query semi-structured data is not limited to information mediators alone. Systems such as W3QS [Konopnicki and Shmueli, September 1988] and WebSQL [Mendelzon *et al.*, 1997] have been developed for structured database like querying of (single) semi-structured Web sources. Besides querying these Web sources directly the purpose of providing a semi-structured query interface may also be to enable warehousing the data in a local text database for more structured querying. The aspects of this thesis dealing with prefetching of data by analyzing the structure of Web sources and costs for various queries could well be applied to optimizing such systems as well.

Both W3QS and Web SQL provide a high level SQL like query language for Web documents and also have extended database query cost estimation techniques to Web sources that can be queried in a structured manner. Thus our techniques for prefetching by source structure analysis could be applied to predetermining expensive queries in these

systems also and prefetching data to main local "views" to improve query response time in such systems. Also our general approach to handling updates at the sources could be applied to keeping these local views consistent.

Reference List

[Abiteboul *et al.*, 1998] Serge Abiteboul, Jason McHugh, Michael Rys, Vasilis Vassalos, and Janet L. Weiner. Incremental maintenance for materialized views over semistructured data. Submitted to VLDB-98, 1998.

[Adali *et al.*, 1997] Sibel Adali, Kasim Seluck Candan, Yannis Papakonstantinou, and V.S.Subrahmanian. Query caching and optimization in distributed mediator systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Tucson, AZ, 1997.

[Agrawal *et al.*, 1993] R. Agrawal, T. Imielinski, and A. Swami. Mining associations between sets of items in massive databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Washington D.C., 1993.

[Ambite and Knoblock, 1998] Jose Luis Ambite and Craig A. Knoblock. Flexible and scalable query planning in distributed and heterogeneous environments. In *Fourth International Conference on Artifical Intelligence Planning Systems*, Pittsburgh, PA, 1998.

[Ambite *et al.*, 1998] Jose Luis Ambite, Craig A. Knoblock, Ion Muslea, and Andrew Philpot. Compiling source descriptions for efficient and flexible information integration. Available at http://www.isi.edu/ knoblock/, 1998.

[Arens and Knoblock, 1994] Yigal Arens and Craig A. Knoblock. Intelligent caching: Selecting, representing, and reusing data in an information server. In *Proceedings of the Third International Conference on Information and Knowledge Management*, Gaithersburg, MD, 1994.

[Arens *et al.*, 1996] Yigal Arens, Craig A. Knoblock, and Wei-Min Shen. Query reformulation for dynamic information integration. *Journal of Intelligent Information Systems, Special Issue on Intelligent Information Integration*, 6(2/3):99–130, 1996.

[Ashish and Knoblock, 1997] Naveen Ashish and Craig A. Knoblock. Semi-automatic wrapper generation for internet information sources. In *Proccedings of the Second IFCIS International Conference on Cooperative Information Systems (CoopIS)*, Charleston, SC, 1997.

[Ashish *et al.*, 1998] Naveen Ashish, Craig A. Knoblock, and Cyrus Shahabi. Intelligent caching for information mediators: A kr based approach. In *Knowledge Representation meets Databases (KRDB)*, Seattle, WA, 1998.

[Ashish *et al.*, 1999] Naveen Ashish, Craig A. Knoblock, and Cyrus Shahabi. Selectively materializing data in mediators by analyzing user queries. In *Fourth International Conference on Cooperative Information Systems (CoopIS)*, Edinburgh, Scotland, September 1999.

[Barish *et al.*, 1999] Greg Barish, Craig A. Knoblock, Yi-Shin Chen, Steven Minton, Andrew Philpot, and Cyrus Shahabi. Theaterloc: A case study in information integration. In *IJCAI Workshop on Intelligent Information Integration*, Stockholm, Sweden, 1999.

[Bayardo *et al.*, 1996] R. Bayardo, W. Bohrer, R. Brice, A. Cichocki, G. Fowler, A. Helal, V. Kashyap, T. Ksiezyk, G. Martin, M. Nodine, M. Rashid, M. Rusinkiewicz, R. Shea, C. Unnikrishnan, A. Unruh, and D. Woelk. Semantic integration of information in open and dynamic environments. Technical Report MCC-INSL-088-96, MCC, Austin, Texas, 1996.

[Bright *et al.*, 1999] L. Bright, L. Raschid, V. Zadorozhny, and T.Zhan. Learning response time for web sources: A comparison of a web based prediction tool (webpt) and a neural network. In *Fourth IFCIS Conference on Cooperative Information Systems (CoopIS)*, Edinburgh, Scotland, 1999.

[Chankunthod *et al.*, 1995] Anawat Chankunthod, Peter B. Danzig, Chuck Neerdaels, Michael F. Schwartz, and Kurt J. Worrell. A hierarchical internet object cache. Technical Report 95-611, Computer Science Department, University of Southern California, Los Angeles, CA, 1995.

[Chawathe *et al.*, 1996] Sudarshan Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. Change detection in hierarchically structured information. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, Montreal, Quebec, Canada, 1996.

[Dar *et al.*, 1996] Shaul Dar, Michael J. Franklin, Bjorn T. Jonsson, Divesh Srivastava, and Michael Tan. Semantic data caching and replacement. In *Proceedings of the 22nd VLDB Conference*, Mumbai(Bombay), India, 1996.

[Etzioni and Weld, 1994] Oren Etzioni and Daniel S. Weld. A softbot-based interface to the Internet. *Communications of the ACM*, 37(7), 1994.

[Finin *et al.*, in press] Tim Finin, Yannis Labrou, and James Mayfield. KQML as an agent communication language. In J. Bradshaw, editor, *Software Agents*. AAAI/MIT Press, Menlo Park, CA, in press.

[Genesereth *et al.*, 1997] Michael Genesereth, Arthur Keller, and Oliver Duschka. Infomaster: An information integration system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Tucson, AZ, 1997.

[Goni *et al.*, 1997] Alfredo Goni, Arantza Illarramendi, Eduardo Mena, and Jose Miguel Blanco. An optimal cache for a federated database system. *Journal of Intelligent Information Systems*, 1(34), 1997.

[Gupta and Mumick, 1998] Himanshu Gupta and Inderpal Singh Mumick. Selection of views to materialize under a maintenance cost constraint. Submitted to VLDB-98, 1998.

[Gupta *et al.*, 1993] Ashish Gupta, Inderpal Singh Mumick, and V.S.Subrahmanian. Maintaining views incrementally. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, Washington, DC, 1993.

[Haas *et al.*, 1997] Laura Haas, Donald Kossmann, Edward Wimmers, and Jun Yang. Optimizing queries across diverse data sources. In *Proceedings of the 23 International Conference on Very Large Databases*, Athens, Greece, 1997.

[Hammer *et al.*, 1995] Joachim Hammer, Hector Garcia-Molina, Kelly Ireland, Yannis Papakonstantinou, Jeffrey Ullman, and Jennifer Widom. Information translation, mediation, and mosaic-based browsing in the tsimmis system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, San Jose, CA, 1995.

[Hyafil and R.L.Rivest, May 1976] H. Hyafil and R.L.Rivest. Construction of optimal binary decision trees is np complete. *Information Processing Letters*, 5:15–17, May, 1976.

[inf, March1997] Informix universal server, version 9.1. Technical report, Informix Press, Menlo Park, CA, March,1997.

[Ives *et al.*, 1999] Zachary Ives, Daniela Florescu, Marc Friedman, Alon Levy, and Dan Weld. An adaptive query execution engine for data integration. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Philadelphia, PA, June 1999.

[Keller and Basu, 1996] Arthur M. Keller and Julie Basu. A predicate–based caching scheme for client–server database architectures. *The VLDB Journal*, 5(2):35–47, 1996.

[Kirk *et al.*, 1995] Thomas Kirk, Alon Y. Levy, Yehoshua Sagiv, and Divesh Srivastava. The information manifold. In *Working Notes of the AAAI Spring Symposium on Information Gathering in Heterogeneous, Distributed Environments*, Technical Report SS-95-08, AAAI Press, Menlo Park, CA, 1995.

[Knoblock *et al.*, 1998a] Craig Knoblock, Steven Minton, Jose-Luis Ambite, Naveen Ashish, Pragnesh J. Modi, Ion Muslea, Andrew Philpot, and Sheila Tejada. Modeling web sources for information integration. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, Madison, WI, 1998.

[Knoblock *et al.*, 1998b] Craig A. Knoblock, Steven Minton, Jose-Luis Ambite, Naveen Ashish, Pragnesh J. Modi, Ion Muslea, Andrew Philpot, and Sheila Tejada. Modeling web sources for information integration. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI)*, Madison, WI, 1998.

[Konopnicki and Shmueli, September 1988] David Konopnicki and Oded Shmueli. Www information gathering: The w3ql language and the w3qs system. *ACM Transactions on Database Systems*, September, 1988.

[Levy *et al.*, 1995] Alon Levy, Alberto Mendelzon, Yehoshua Sagiv, and Divesh Srivastava. Answering queries using views. In *Proceedings of the 14th ACM SIGACT-SIGMOD-SIGARTSymposium on Principles of Database Systems*, San Jose, CA, 1995.

[MacGregor *et al.*,]
Robert MacGregor, Hans Chalupsky, and Eric Melz. Powerloom manual. Available online at http://www.isi.edu/isd/LOOM/PowerLoom/documentation/manual.html.

[MacGregor, 1988] Robert MacGregor. A deductive pattern matcher. In *Proceedings of AAAI-88, The National Conference on Artificial Intelligence*, St.Paul, MN, 1988.

[Mendelzon *et al.*, 1997] Albert Mendelzon, George A. Mihalia, and Tova Milo. Querying the worl wide web. *Journal of Digital Libraries*, 1(1):68–88, 1997.

[Muslea *et al.*, 1998] Ion Muslea, Steven Minton, and Craig A. Knoblock. A hierarchical approach to wrapper induction. In *Proceedings of the Third Internation Conference on Autonomous Agents*, Seattle, WA, 1998.

[Quass *et al.*, 1996] Dallan Quass, Ashish Gupta, Inderpal Singh Mumick, and Jennifer Widom. Making views self-maintainable for data warehousing. In *Parallel and Distributed Information Systems (PDIS)*, Miami Beach, Florida, 1996.

[S.J.Wan *et al.*, June 1988] S.J.Wan, S.K.M Wong, and P. Prusinkiewicz. An algorithm for multi dimensional data clustering. *ACM Transactions on Mathematical Software*, 14(2):153–162, June, 1988.

[Srikant and Agrawal, 1995] R. Srikant and R. Agrawal. Mining generalized association rules. In *Proceedings of the 21st International Conference on Very Large Databases*, Zurich, Switzerland, 1995.

[Tomasic *et al.*, 1997] Anthony Tomasic, Louiqa Raschid, and Patrick Valduriez. A data model and query processing techniques for scaling access to distributed heterogeneous databases in disco. In *Invited paper in the IEEE Transactions on Computers, special issue on Distributed Computing Systems*, 1997.

[Wiederhold, 1992] Gio Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, March 1992.

[Yang *et al.*, 1997] Jian Yang, Kamalakar Karlapalem, and Qing Li. Algorithms for materialized view design in data warehousing environment. In *Proceedings of the 23 International Conference on Very Large Databases*, Athens, Greece, 1997.