MEDIATED QUERY PROCESSING
OVER AUTONOMOUS DATA SOURCES

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

By
Ramana Yerneni
August 2001

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

Héctor García-Molina
(Principal Adviser)

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

Jeffrey D. Ullman

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

Yannis Papakonstantinou

Approved for the University Committee on Graduate Studies:

Abstract

When processing queries over autonomous data sources, users face many challenges. We focus our attention on the problems encountered due to the limited and diverse query-processing capabilities of data sources and the need to integrate data from a large number of sources in order to answer user queries. Mediators are developed to alleviate these problems, as they provide extended query-processing capabilities and integrated views across data sources. In this thesis, we discuss the techniques we have developed to enable mediators to overcome the challenges of processing queries over large sets of autonomous sources that have limited and diverse capabilities.

We develop languages to describe query-processing capabilities. We present algorithms used by mediators to support powerful query interfaces to data sources by translating the user queries into a sequence of simpler sources queries and postprocessing operations at the mediators. We discuss the complexity of the problem of query planning at the mediators when processing queries over large join views and develop good optimization algorithms. We identify a class of queries that often occur in contexts involving autonomous data sources, namely *fusion queries*. Conventional query-optimization techniques do not scale well to process fusion queries, and hence we develop new techniques for fusion-query optimization. We consider the framework of a network of mediators working together to process user queries and in this context we present algorithms to compute the query capabilities of mediators based on the capabilities of data sources and other mediators they rely on. We also discuss ways in which the query capabilities of data sources and mediators can be expressed concisely to facilitate a simpler user experience and more efficient query processing.

To my parents, wife and daughter

Acknowledgments

I would only be restating the obvious when I say that Prof. Hector Garcia-Molina is a great advisor. He was instrumental in my joining the program at Stanford and right from day one has been a source of tremendous help and encouragement. I have learnt much from him and I will continue to learn much the rest of my life by recounting the ideas he has discussed while helping me in many crucial stages of my stay at Stanford. I consider myself truly fortunate to have him as my advisor and thank him sincerely for the opportunity to work under his guidance.

Prof. Jeffrey D. Ullman has helped me tremendously in making steady progress in my Ph.D. work. I benefited significantly from his counsel and guidance in many important situations and I thank him for his kindness and help. I enjoyed the rewarding experience of having worked closely with him on interesting problems. He has helped not only in the development of the ideas and solutions, but also in the careful presentation of them.

Many others at Stanford, faculty as well as students, have helped in significant ways. Of particular mention is Prof. Yannis Papakonstantinou. He helped me work on my first research problem at Stanford and was instrumental in publishing my first research paper at Stanford. He has also helped me understand the process of identifying research problems, crafting useful solutions and presenting the solutions through research papers. Finally, he has graciously agreed to review my thesis inspite of a very busy schedule.

I am also indebted to the many people at Stanford that worked closely with me on research problems, co-authored research papers and built research prototypes with me. Some of the key people I would like to thank in this respect are: Prof. Serge Abiteboul, Joachim Hammer, Wilburt Labio, Murty Valiveti, Chen Li, Vasilis Vassalos and Svetlozar Nestorov.

My wife, Lakshmi, has been a constant source of encouragement and support throughout the process of applying to Stanford, joining Stanford, and working on my Ph.D. We had a great time living on the campus during my Ph.D. stint, and we had the wonderful experience of becoming parents midway through. Our daughter, Anita, has thoroughly enjoyed the Stanford experience. I also thank my parents and my sisters for motivating me constantly to strive for higher goals in my life. A big part of the reason for my pursuing Ph.D. studies at Stanford is their encouragement and support.

Contents

Abstract **iv**

Acknowledgments **vii**

1 Introduction **1**

1.1 Limited Capabilities of Sources 2
1.2 Extending Capabilities through Mediators 3
1.3 Integrated Views across Multiple Data Sources 5
1.4 Fusion Queries 7
1.5 Mediator Capabilities 9
1.6 Concise Capability Descriptions 10
1.7 Thesis Outline 11

2 Extending Capabilities of Data Sources **13**

2.1 Overview 13
2.2 Preliminaries 16
2.2.1 Notation 16
2.2.2 Query Plans 17
2.2.3 Describing Source Capabilities 18
2.3 A Modular Scheme 20
2.3.1 Rewrite Module 21
2.3.2 Mark Module 21
2.3.3 Generate Module 22
2.4 GenCompact 25
2.4.1 Rewrite Module 25

2.4.2 Cost Model 27
2.4.3 Pruning Rules 28
2.4.4 Plan Generation Module 29
2.4.5 Integrated Plan Generator 29
2.4.6 Processing an $\vee$ Node 30
2.4.7 Processing an $\wedge$ Node 33
2.5 Performance Evaluation 36
2.5.1 Efficiency of *GenCompact* 37
2.5.2 Quality of *GenCompact* Plans 39
2.6 Discussion 40
2.7 Related Work 42
2.8 Chapter Summary 43

3 Optimizing Large Join Queries **44**
3.1 Overview 44
3.1.1 Capabilities-Based Plan Generation 44
3.2 Framework 48
3.2.1 Cost Model 49
3.2.2 Subgoals and Sequences 50
3.2.3 The Space of Query Plans 51
3.2.4 Complexity of the Problem 53
3.3 A Greedy Approach 55
3.3.1 Quality of Plans Generated by CHAIN 55
3.4 A Clustering Approach 59
3.4.1 Feasible Plan Generation 61
3.4.2 Comparison with CHAIN 62
3.5 Performance Analysis 63
3.5.1 Simulation Environment 64
3.5.2 Finding Optimal Plans 64
3.5.3 Margin of Error from Optimal Plans 65
3.6 Related Work 66
3.7 Chapter Summary 67

4 Fusion-Query Optimization **68**

4.1 Overview . . . 68

4.2 Framework . . . 73

4.2.1 The Sources . . . 73

4.2.2 Fusion Queries . . . 74

4.2.3 Simple Plans . . . 75

4.2.4 Cost Model . . . 76

4.2.5 Important Classes of Simple Plans . . . 76

4.3 Finding Optimal Simple Plans . . . 78

4.4 Optimality of Semijoin-Adaptive Plans . . . 82

4.5 Postoptimization . . . 86

4.5.1 SJA+ . . . 88

4.5.2 Other Postoptimization Techniques . . . 89

4.6 Performance Evaluation . . . 90

4.6.1 Comparing the Optimization Algorithms . . . 92

4.6.2 Correlated Conditions . . . 94

4.7 Fusion Queries in Existing Optimizers . . . 96

4.8 Chapter Summary . . . 100

5 Computing Mediator Capabilities **102**

5.1 Overview . . . 102

5.2 Framework . . . 104

5.2.1 Mediator Views . . . 104

5.2.2 Query Templates . . . 105

5.3 Templates for Simple Mediators . . . 106

5.3.1 Union Views . . . 107

5.3.2 Join Views . . . 109

5.3.3 Selection Views . . . 110

5.3.4 Projection Views . . . 110

5.3.5 Templates for Graph of Views . . . 111

5.4 Advanced Query-Processing Techniques . . . 111

5.4.1 Union Views . . . 112

5.4.2 Join Views . . . 114

5.4.3 Selection and Projection Views 115
5.5 Dynamic Mediators 117
5.5.1 Conservative and Liberal Templates 118
5.5.2 Computing Liberal Templates 119
5.5.3 Content Information and Query Answerability 120
5.6 Output Restrictions on Attributes 121
5.6.1 Template Computation for Simple Mediators 122
5.6.2 Template Computation for Complex Mediators 123
5.6.3 Template Computation for Dynamic Mediators 127
5.7 A Case Study 128
5.8 Related Work 133
5.9 Chapter Summary 134

6 Minimization of Template Sets **135**
6.1 Overview 135
6.1.1 Designing Source Interfaces 135
6.1.2 Concise Source Interfaces 136
6.2 The Template-Set Minimization Problem 138
6.3 Template Subsumption 140
6.3.1 Algorithm SIMPLE 141
6.4 Irredundant Template Sets 145
6.4.1 Detecting Template Redundancy 146
6.4.2 Eliminating Redundant Templates 149
6.5 Template Difference 150
6.6 Template Coalescing 154
6.7 Related Work 158
6.8 Chapter Summary 159

7 The TSIMMIS System **160**
7.1 Overview 160
7.2 The TSIMMIS Mediator 161
7.2.1 View Expander 161
7.2.2 Plan Generator 163
7.2.3 Source Capabilities 164

7.2.4 Matcher 165
7.2.5 Sequencer 165
7.2.6 Optimizer 166
7.2.7 Execution Engine 166
7.3 The TSIMMIS Wrappers 166
7.3.1 Query Parser 167
7.3.2 Native Component 169
7.3.3 Postprocessing Engine 170
7.4 Toolkits 170
7.5 Chapter Summary 171

8 Conclusion **173**

Bibliography **176**

List of Tables

3.1 Database Instance: R, S and T. 57
3.2 Database Instance: R, S, T, U. 62

5.1 Mapping Function for Combining Adornments from Two Templates. 108
5.2 Mapping Function for Union with Postprocessing. 113
5.3 Mapping Function for Join by Passing Bindings. 114
5.4 Mapping Function for Selection with Postprocessing. 116
5.5 Mapping Function for Projection with Postprocessing. 116
5.6 Mapping Function for Liberal Join-View Templates. 119
5.7 Simple Union and Join with Output Restrictions. 122
5.8 Union with Postprocessing and Output Restrictions. 123
5.9 Join with Postprocessing and Output Restrictions. 125
5.10 Selection with Postprocessing and Output Restrictions. 125
5.11 Projection with Postprocessing and Output Restrictions. 127
5.12 Liberal Templates of Join Views with Output Restrictions. 127
5.13 Templates of Amazon. 129
5.14 Templates of BarnesAndNoble. 130
5.15 Mediator Templates before Minimization. 130
5.16 Mediator Templates after Minimization. 131

6.1 Adornment Restrictiveness Checking. 140
6.2 Computing Difference of Two Adornments: DiffAdorn. 152
6.3 Combining Adornments from Coalesced Templates. 157

7.1 Matcher Result 165

List of Figures

2.1 Condition Tree. 16
2.2 Query Tree. 16
2.3 Query Tree. 16
2.4 *GenModular* Scheme for Generating Query Plans. 20
2.5 Algorithm EPG. 24
2.6 Algorithm IPG. 30
2.7 $\vee$ Node Processing of Algorithm IPG. 32
2.8 $\wedge$ Node Processing of Algorithm IPG. 35
2.9 Efficiency of *GenCompact* in Processing "Sensible" Queries. 38
2.10 Efficiency of *GenCompact* in Processing a Random Query. 39
2.11 Comparing Quality of Plans. 40

3.1 Algorithm *CHAIN*. 56
3.2 Proof of Lemma 3.3.2. 58
3.3 Algorithm *PARTITION*. 60
3.4 Fraction of Non-optimal Plans. 64
3.5 Average Cost. 65
3.6 Worst-case Miss. 66

4.1 DMV Example. 69
4.2 Three Simple Plans. 76
4.3 Relationship between the Subsets of Simple Plans. 79
4.4 The SJ Algorithm. 80
4.5 The SJA Algorithm. 81
4.6 Postoptimization. 87
4.7 Varying Number of Sources. 93

4.8 Varying Number of Conditions. 94
4.9 Postoptimization Improvements. 95
4.10 Correlated Conditions. 96
4.11 Distributing Join over Union. 98
4.12 Handling Unions Uniformly. 99

6.1 Algorithm *SUBSUME*. 141
6.2 Algorithm *SIMPLE*. 142
6.3 Algorithm *ATOMIZE*. 147
6.4 Algorithm *REDUNDANT*. 148
6.5 Algorithm *SPLITCHECK*. 149
6.6 Algorithm *DIFFERENCE*. 152
6.7 Algorithm *DIFFCHECK*. 153
6.8 Algorithm *COALESCE*. 156
6.9 Algorithm *CONDENSE*. 157

7.1 The TSIMMIS Architecture. 160
7.2 Mediator Architecture. 162
7.3 Wrapper Components. 167

Chapter 1

Introduction

With the advent of the Web, information is available more easily from a variety of data sources. When we want to purchase a book, there are many Web sources that present information about the desired book. When we want to gather information about a company in the context of exploring an investment opportunity, the relevant information can be obtained from a large number of Web sources that present financial, corporate, product, and market information about the company.

Data sources on the Web, unlike traditional databases, are evolving in an autonomous manner. A typical data source on the Web is much more concerned about the data it presents, and the unique ways in which it presents its data, than about setting up its content or presentation in a manner that is compatible with other related data sources. The autonomous nature of Web sources leads to a relatively arbitrary data fragmentation and heterogeneous query interfaces at the various data sources. Navigating the different query interfaces and harnessing the data from across multiple relevant sources is a big challenge for users. Mediators facilitate these tasks and make it easy for users to access the data conveniently and efficiently.

In this thesis, we study the problems faced by users in accessing relevant data from a set of autonomous sources. We describe the solutions offered by mediators to solve these problems and identify the specific challenges faced by mediators in solving these problems. We present the techniques and algorithms we have developed to help mediators overcome these challenges and provide their customers with efficient and convenient access to data across autonomous sources. Even though we discuss many example scenarios involving Web sources to illustrate the problems and solutions we studied in this thesis, these problems and solutions are also applicable in the Intranet context. For instance, query processing over a corporate Intranet has to deal with the autonomous sources of data managed by the different organizations of the corporation.

In the rest of this chapter, we illustrate the problems solved in the thesis through a variety of simple examples. The essential ideas are discussed informally in this chapter, while subsequent chapters present the necessary details formally. In Section 1.1, we describe how Web sources present limited query capabilities to their users. Next, we discuss how mediators can extend such limited capabilities by using a variety of techniques. In the following section, we deal with scenarios in which mediators process user queries on integrated views that join information from multiple data sources. In Section 1.4, we introduce the notion of a special kind of queries, called *fusion queries*, for which we develop scalable query-optimization techniques. The next two sections describe a framework in which the query capabilities of data sources can be expressed in a simple and elegant manner and the capabilities of mediators can be computed from the capabilities of the sources they mediate. Finally, in Section 1.7, we discuss the outline of the rest of the thesis.

1.1 Limited Capabilities of Sources

Data sources on the Web often have varying and limited query-processing capabilities. Processing queries over such sources poses many challenges as illustrated by the following example.

EXAMPLE 1.1.1 Consider the Web bookstore Amazon.com [85].[1] Users pose queries to this source by filling out a search form. Amazon's search form supports queries based on author, title, subject, ISBN, and publisher attributes. Specifically, the user must fill out at least one of the five attributes and submit the form. In response Amazon will return a list of books with some of these and additional attributes. In particular, it does not return the subject attribute and it returns the price attribute among others.

Many simple queries cannot be posed to Amazon using this search interface. For instance, suppose the user wants to find the list of books written by "Freud" that cost at most \$10. Amazon does not support this query or any query that specifies price conditions.

[1] All features of real sources mentioned in this thesis are as of the time when research was conducted on those sources. The last date when the features were checked is indicated in the bibliography.

Consider another Web bookstore BarnesAndNoble.com [90]. Once again users can pose queries by filling out a search form. BarnesAndNoble's search form [91] allows users to specify at least one of author, title, and keywords attributes, along with optional specification of price, format and other attributes. In particular, BarnesAndNoble can support the query to find all books by "Freud" that cost at most \$10. However, it cannot support many other simple queries like "find all books published by McGraw-Hill." Incidentally, this query is supported by Amazon. □

One might wonder why sources have limited query capabilities. There are many reasons for capability limitations. Chief among them are:

1. Sources may choose to provide simple query interfaces through search forms even if they are actually capable of answering more complex queries.
2. Certain attributes may be hidden for security reasons. Such attributes may not appear in the query results and/or they may not appear in the query-condition specification.
3. Sources may only allow conditions on attributes for which they have indexes so that certain query response-time profiles can be maintained.
4. There may be published *application programming interfaces* (APIs) that are in current use, so they continue to support such interfaces even when they are restrictive.

1.2 Extending Capabilities through Mediators

As illustrated by Example 1.1.1, the limited capabilities of data sources make the processing of even simple queries challenging. Mediators are constructed to overcome these challenges. Users can submit complex queries to mediators. Such complex queries, if posed directly at the data sources, would be rejected by the data sources. Mediators use query-decomposition and postprocessing techniques to support complex queries. These techniques are illustrated by the following examples.

EXAMPLE 1.2.1 Consider the query "find all books written by Freud that cost at most \$10." This query is not allowed by Amazon. A mediator can be built on top of the Amazon data source in such a way that the query can be answered. The mediator can send a different query "find all books written by Freud" to Amazon and then postprocess the results by applying the *filtering* condition to eliminate books that cost more than \$10. Thus, the mediator can allow users to submit more powerful queries than those directly supported by Amazon. □

EXAMPLE 1.2.2 Consider the BarnesAndNoble data source again. Let the desired query be "find all books written by Freud or Jung on the topic of Dreams." Since BarnesAndNoble's search form does not allow the specification of disjunctive conditions, this query cannot be submitted to the data source. A mediator built on top of BarnesAndNoble can support the query in the following manner. The mediator can decompose the given query into two separate queries: "find all books written by Freud on the topic of Dreams" and "find all books written by Jung on the topic of Dreams." Then it can submit the two queries, one at a time, to BarnesAndNoble's search engine and obtain the two sets of results. Finally, it can union the two sets of results and present the combined result to the user. □

Note that in Example 1.2.2, there are many ways the mediator can decompose the user query. Correspondingly, the mediator can consider a variety of *query plans*. For instance, the mediator can consider the following alternative plan: send the query "find all books written by Freud," collect results into the set R_1, filter out books that are not on the topic of "Dreams" to obtain the set R_2, send the query "find all books written by Jung," collect results into the set R_3, filter out books that are not on the topic of "Dreams" to obtain the set R_4, combine the sets R_2 and R_4 to obtain the final result and present the result to the user. One can see that there are additional other plans that the mediator can also consider. The various plans have different performance implications. That is, some plans can be executed faster than others. It is the job of the mediator to consider a variety of plans and pick a good plan.

The challenge faced by the mediator in this situation is that the number of plans to consider grows rapidly as the complexity of the user query increases. Even for simple queries with a few attribute conditions, the mediator needs to consider potentially thousands of plans and choose the plans with good performance characteristics. In Chapter 2, we describe the techniques we have developed in order to enable mediators to efficiently choose good plans for complex user queries.

1.3 Integrated Views across Multiple Data Sources

In the previous section, we discussed how mediators can be employed to extend the sets of supported queries of sources with limited capabilities. There, each user query is targeted to a single source and the mediator for the source allows the user to pose queries that are more complex than what the source might accept.

A very important function of mediators is to provide integrated access to data from multiple sources. Essentially, mediators provide integrated views on which users can pose queries. These queries are then decomposed into queries on the corresponding data sources that are sent to the data sources. The mediators combine the results of the source queries and present the final results to the users. The capability limitations of the data sources pose challenges to mediators when processing queries on their integrated views.

Two of the most common types of views provided by mediators are: *union* and *join*. In the case of a union view, the user query is to be executed at each data source and the result sets are to be unioned. When processing a query submitted on the union view, the mediator can use the techniques described in Section 1.2 to allow powerful queries, as illustrated by the following example.

EXAMPLE 1.3.1 Consider the Amazon and BarnesAndNoble data sources of Example 1.1.1. A mediator can be constructed on top of these two data sources to provide comparison-shopping experience for users. The mediator can provide a union view of the two data sources so that users can find the desired books at both the bookstores by submitting a single query and comparing the corresponding results from both the bookstores.

Let the user submit the query "find all books by Freud that cost at most \$10." The mediator can recognize that this query is not supported by Amazon's data source. Instead, it sends the alternate query "find all books by Freud" to Amazon and obtains the result set R_1. It can postprocess R_1 with respect to the price condition to obtain the result set R_2. For BarnesAndNoble, it can send the user query as is and obtain the result set R_3. Then it can combine the sets R_2 and R_3 to obtain the final result R_4. □

As illustrated by the above example, the challenges posed by the capability limitations of data sources on the processing of user queries on union views is easily handled using the techniques developed in Section 1.2. The case of join views on the other hand poses special challenges that need the development of additional techniques.

EXAMPLE 1.3.2 Consider three data sources that provide information related to movies. Let the first source provide a simple view $R(studio, title)$, let the second source provide the view $S(title, year)$ and let the third source provide the view $T(title, stars)$. For the first source, let the search form require the specification of the *title*, let the form for the second source require the specification of either the *title* or the *year* and let the form for the third source require either the *title* or the *stars* to be specified.

Now consider a mediator built on top of these three sources to provide an integrated view $Movie(studio, title, year, stars)$ that is a join of the three source views. Let the user query the mediator to find information about all the movies made by "Paramount" studio in "1955" starring "Gregory Peck." A simple query-translation process at the mediator may generate the following plan: send the query R("Paramount", $title$) to the first source; send the query S($title$, "1955") to the second source; send the query T($title$, "Gregory Peck") to the third source; join the results of all three queries, on the *title* attribute, to arrive at the answer to the user query. Because of the search-form constraint of the first source, such a query plan cannot be executed. In particular, the query R("Paramount", $title$) is not answerable by the first source. However, the mediator should not reject the query because there are ways to execute the user query over the data sources.

The mediator can first send the query S($title$, "1955") to the second source and the query T($title$, "Gregory Peck") to the third source; join the results based on the *title* values; for each title in the join result, it can get the *studio* information of the movie from the first source and check if it is "Paramount;" if so, it can return the movie information to the user, else it can remove that title from the result of the user query. □

As observed in the above example, when processing queries that join information across data sources with query-capability limitations, the mediator needs to consider different orders of accessing the sources and *passing bindings* from the results of some source queries to others. Some orders lead to feasible query-execution plans while others do not. For instance, in Example 1.3.2, there are a total of six orders involving the three data sources. Only four of these six orders lead to feasible query plans, those that visit either the second or the third source first.

There could also be significant differences in the efficiency of query execution among the various source-query orders. Thus, the mediator needs to not only find a feasible order but also ensure that the chosen order is an efficient one. Since the number of orders of source queries grows exponentially with the number of sources to be queried, when processing user queries across large numbers of sources, the mediator's task of finding an efficient feasible query plan becomes quite challenging. In Chapter 3, we describe special techniques and algorithms that we have developed to help mediators find good query plans for large join queries in an efficient manner.

1.4 Fusion Queries

As we noted earlier, data sources on the Web are autonomously evolving. Accordingly, the data distribution across these sources is not stipulated in any traditional "top-down" manner. Fragments of related information about an entity are available from multiple sources. Mediators need to offer users an integrated view of the data so that they can pose queries on the simpler view of the data they are presented with.

EXAMPLE 1.4.1 Consider a set of autonomous sources offering information about the automobile-driving activities of a person. These sources could be the databases managed by the automobile regulatory departments of various states in the country.

Let us consider the data related to the traffic violations of automobile drivers. A person from "California" could have driven in many different states and may have been cited for various violations. Let us assume that each state authority records the traffic citations issued within its jurisdiction. Such data may not be transferred to and consolidated in the database of the "home" state of each driver. In the resulting scenario, the driving portfolio of any driver is potentially dispersed across the data sources maintained by the different states. If we are trying to find drivers that fit a particular profile, it is quite cumbersome to access the different data sources and combine the results of all the inquiries to arrive at the correct answer.

In such a scenario, mediators provide users with a convenient *fusion* view on which the users operate. Suppose we are looking for drivers that have *speeding* as well as *drunk-driving* violations within the past year. Without the mediator providing us with the fusion view, we have to collect the lists of drivers at each data source having either the speeding violation or the drunk-driving violation, and then correlate the results from all the source queries. A driver may have a speeding violation recorded at one source and a drunk-driving violation recorded at another source. On the other hand, there can be drivers that have both kinds of violations present in the same data source. If a mediator presents us with the fusion view, we can avoid visiting all the sites and correlating their results by simply posing the query to the mediator, indicating that we are looking for drivers with speeding and drunk-driving violations. The mediator constructs an efficient query plan involving the relevant source queries and postprocessing operations at the mediator in order to combine the source-query results to obtain the answer to our query. □

Fusion views appear to be very similar to union views and join views. In fact, a fusion view is merely a combination of the union and join views. If we consider the data fragments stored at the various data sources as records in tables, a fusion view starts out with the union of all the tables. Of course, in this union, each entity has multiple records. For instance, in the case of the above example, the union of all the tables can have multiple records for each driver. When the user submits a list of criteria to be satisfied by a driver, the query is translated into a multi-way self-join of this union (the join attribute being the drivers' license or identifier), with the specific conditions being met by the constituent records. That is, a query on the fusion view, we call such a query a *fusion query*, can be viewed as one that involves a multi-way self-join over a union of a set of tables (the join attribute(s) being the key or the entity identifier).

The special challenge faced by mediators in processing fusion queries is that the number of sources participating in a fusion query tends to be large and the amount of data at each source can also be large. It is not a good strategy to simply "upload" the entire data from the various sources and materialize the view at the mediator. Also, at query-execution time, it is not usually efficient to transfer a lot of data from a large number of sources. Thus, the mediator is faced with the challenge of minimizing data transfers across sources while at the same time providing a seamlessly-integrated fusion view to the users.

Contemporary systems do not handle fusion queries adequately because such queries occur rarely in traditional database settings where top-down designs for data distribution ensure that small numbers of sources need to be combined to process distributed queries. Moreover, based on the knowledge of the data characteristics (e.g., all data about a California driver will be in the California data source, or certain traffic violations are only recorded in a subset of the data sources), the query processor can increase the efficiency of the query execution significantly. In the context of Web sources, data is fragmented and distributed in an unpredictable manner and we cannot make assumptions as easily about the characteristics of data sets when autonomous sources are involved. The conventional techniques, like pushing the join operations through the union operations in query trees and trying to optimize the query execution by looking for common subexpressions to eliminate, do not scale well for fusion-query processing.

In Chapter 4, we present the techniques we have developed to aid mediators in efficient fusion-query processing. We describe our algorithms and present theoretical as well as simulation-based performance characteristics of these algorithms.

1.5 Mediator Capabilities

Data sources with limited capabilities inform mediators about their supported-query sets. The capability limitations of sources translate into the corresponding limitations of the mediator in terms of restricting the set of answerable queries on the views exported by the mediators. Just as sources inform their users about their query capabilities, mediators should also describe to their users which queries they can support, so that users may know which queries to submit. In contemporary systems such as Information Manifold [37, 40], Garlic [9, 28], and DISCO [35, 71], sources express their capabilities through a variety of mechanisms – query templates, capability records, and simple capability-description grammars. However, none of these systems computes the query-capabilities of mediators based on their source capabilities. Not having the mediator capabilities readily available makes it difficult to treat mediators as sources for other mediators. Furthermore, users have a difficult time understanding the set of supported mediator queries in these systems. Consequently, users must endure a frustrating trial-and-error approach, submitting queries that are rejected until finally hitting upon a query that is answered by the mediator.

Computing mediator capabilities can be simple in some cases. We may be able to combine all the restrictions of the sources in an easy manner to arrive at the capabilities of the mediator. The following example illustrates such a simple case.

EXAMPLE 1.5.1 Consider the Web bookstores Amazon and BarnesAndNoble of Example 1.1.1. Recall that the search form of Amazon allows search based on author, title, subject, ISBN, and publisher attributes. Specifically, the user must fill out at least one of the five attributes. BarnesAndNoble's search form allows users to specify at least one of author, title, and keywords attributes, along with optional specification of price, format and other attributes.

Now consider a mediator that provides a union view over the two data sources (like the one in Example 1.2.2). The mediator can publish a search form that allows users to submit queries by specifying at least one of author and title. Essentially, the capabilities of the mediator can be computed as the common capabilities of the sources in its union view. That is, any user query that specifies either the author or the title is supported by both the sources. So, the mediator can support such a query. □

In many cases, the computation of mediator capabilities is more complicated due to the rich variety of source-capability limitations and the host of techniques mediators employ in processing queries. For instance, as we saw in Section 1.2, by using special techniques to postprocess the results of queries on limited-capability data sources, mediators can support much larger sets of queries. Thus, the capabilities we compute for mediators can be "stronger" than those of the underlying sources. For instance, the Amazon data source does not allow price conditions while BarnesAndNoble does not allow publisher conditions. By using postprocessing techniques the mediator can allow its users to specify price conditions and/or publisher conditions on the union view.

In Chapter 5, we present algorithms that we have developed to compute the mediator capabilities automatically. Our algorithms take into consideration various classes of mediators that employ a variety of techniques to extend the set of queries supported by their sources.

1.6 Concise Capability Descriptions

Templates are often used to describe query capabilities of sources. A template indicates the restrictions on attribute specification in queries to the source (e.g., some attributes are required while others are optional). These restrictions are specified as attribute *adornments* in the templates. The capabilities of mediators are computed from those of its data sources. The number of templates computed for the mediator views can be very large when compared with the number of templates of the source views that are input to the computation. For instance, in the case of a mediator view that is a union of n source views with k templates each, we can end up with as many as k^n templates.

Besides the space it takes to store the mediator templates, a more serious concern here is that the query-capability description of the mediator may become so large as to make it difficult to ascertain whether a given query is answerable. A user trying to figure out if a candidate query should be posed to a mediator would like a succinct specification of the mediator query capabilities. Also, another mediator or application relying on the mediator with limited capabilities needs to take into consideration its large set of templates when finding optimal (feasible) query plans.

Fortunately, we may be able to reduce the size of the capability description significantly, based on the concept of minimizing template sets. Given a set of templates we can try to find another set that is equivalent to the given set but at the same time containing fewer templates.

The problem of minimizing template sets is a formidable one. In Chapter 6, we study the complexity of this problem and show that the general case of this problem is intractable. We identify many practical scenarios in which we construct efficient algorithms to solve the problem optimally. We also present good approximation algorithms for the general case of the problem.

1.7 Thesis Outline

In this thesis, we study the challenges faced in processing queries over autonomous data sources. In particular, we focus on two kinds of challenges. The first kind is based on the limited and diverse capabilities of the autonomous sources. The second kind is based on the distribution of data in an unpredictable manner across large numbers of sources. Mediators built on top of autonomous data sources facilitate the processing of powerful queries over the data sources by providing integrated views of data and extended capabilities. In this thesis, we describe the challenges posed by the autonomous data sources and present techniques we have developed to enable mediators to overcome these challenges.

In Chapter 2, we describe how mediators extend the capabilities of data sources through the application of postprocessing and query-decomposition techniques. We identify the query-optimization challenges faced by mediators in order to find the most efficient ways to execute complex user queries on sources with limited capabilities. We then present algorithms we have developed to overcome these challenges and enable mediators to efficiently find good plans for complex queries.

We discuss the technique of passing bindings across sources, in Chapter 3, when dealing with the processing of join queries over data sources with query-capability limitations. Once again, the key challenge for the mediator is to efficiently find good feasible plans in a large space of plans. We study the complexity of the optimization problem and show that the problem is $\mathcal{NP}$-hard. We present the approximation algorithms we have developed and show that they have theoretical guarantees in terms of *quality of plans* and we also study their practical performance.

In Chapter 4, we present the notion of fusion queries in the context of autonomous data sources over the Web. We define the problem of fusion-query optimization faced by mediators integrating data across large numbers of sources. The problem in general is intractable. We present techniques we have developed to identify special classes of query plans for processing fusion queries and describe algorithms to find optimal plans in those classes. We also discuss theoretical as well as practical performance of these optimal plans relative to each other and with respect to optimal plans in the general space of all query plans for fusion queries.

Chapter 5 describes the framework we have developed to formally describe the query capabilities of data sources and how we can compute the capabilities of mediators based on the capabilities of the data sources they mediate. We present algorithms to help compute the mediator capabilities for different kinds of mediators that employ a variety of techniques (e.g., postprocessing and passing bindings) to extend the sets of queries they support. We also discuss the notion of dynamic mediators whose capabilities are dependent on the data existing in the sources at the time of query execution.

In Chapter 6, we define and study the problem of template-set minimization. Solving this problem enables us to compute concise capability descriptions of data sources and mediators. We show that the general case of the problem is intractable. We present algorithms that optimally solve useful special cases of the problem that occur in practical scenarios. We also develop techniques that yield good approximate solutions for the general case of the problem.

We have developed a research prototype named TSIMMIS at Stanford University to study various challenges in building mediation systems and experiment with our solutions to overcome these challenges. We present the TSIMMIS system in Chapter 7. We briefly describe the overall architecture of this research prototype with particular emphasis on issues surrounding the description of source capabilities, query-capability extension through postprocessing of source-query results, and data integration across multiple sources through mediators.

Chapter 2

Extending Capabilities of Data Sources

2.1 Overview

Data sources over the Web have a wide range of query-processing capabilities. In particular, many sources provide a single-table view, and allow only limited types of selection queries. The limited query capabilities of sources introduce interesting query-processing challenges, as illustrated by the following examples.

EXAMPLE 2.1.1 Consider the Web bookstore Amazon.com [85]. Suppose one wants to look for books written by "Sigmund Freud" or "Carl Jung" on the topic of "Dreams." The search form provided by BarnesAndNoble does not allow us to search for two authors at once, so a good plan is to break up the query into two. Thus, we can first look for books satisfying ($author$ = "*Sigmund Freud*" $\wedge$ *title contains* "*Dreams*"); and then look for books satisfying ($author$ = "*Carl Jung*" $\wedge$ *title contains* "*Dreams*"). The results of the two queries can then be unioned to obtain the answer to the original query.

Most current query-processing systems would be unable to come up with a good plan for this simple example query. Many systems simply assume that sources have full relational capabilities, and would try sending the full unsupported query to the Amazon source. Furthermore, systems that do take into account source capabilities, only consider limited options. For example, in a system like Garlic [28], query conditions are always processed in conjunctive normal form (CNF). Therefore, our query condition would be transformed to $((author = \text{“}Sigmund\ Freud\text{”} \vee author = \text{“}Carl\ Jung\text{”}) \wedge (title\ contains\ \text{“}Dreams\text{”}))$. Garlic realizes that the first clause cannot be sent to Amazon, but that the second one can be. It thus sends the second clause and applies the first one itself. This plan is valid, but extracts many more entries from Amazon than the two-query plan. Thus, if we are concerned about the amount of data retrieved, the Garlic plan is not very good.

Of course, a query-processing system that uses disjunctive normal form (DNF) can come up with our plan. However, there are many examples where CNF instead of DNF is the “right” choice, and there are many other examples where neither CNF nor DNF is a good strategy, as illustrated by the example below.

The key point is that current systems either ignore source capabilities or only consider limited types of plans, leading to query plans that may be infeasible or inefficient. □

EXAMPLE 2.1.2 Consider an automobile-purchasing site like AutoConnect.com [89]. One can pose queries to such a source regarding cars for sale, using various attributes. Suppose we are looking for information on midsize or compact sedans. In particular, we are interested in Toyotas under \$20,000, and BMWs under \$40,000. The query condition in this case is: $(style = \text{“}sedan\text{”} \wedge (size = \text{“}compact\text{”} \vee size = \text{“}midsize\text{”}) \wedge ((make = \text{“}Toyota\text{”} \wedge price <= 20000) \vee (make = \text{“}BMW\text{”} \wedge price <= 40000)))$. Let the data source have a query form that allows users to specify single values for $style$, $make$ and $price$ along with a list of values for $size$. Hence, our query condition cannot be supported by the data source. However, we can break it up into two conditions: $(style = \text{“}sedan\text{”} \wedge make = \text{“}Toyota\text{”} \wedge price <= 20000 \wedge (size = \text{“}compact\text{”} \vee size = \text{“}midsize\text{”}))$ and the corresponding one for BMWs. We can send these two queries to the data source, and union their results to obtain the answer to the original query.

In this example, both DNF and CNF query-processing systems produce less desirable plans. In a DNF system such as Information Manifold [41], the user query is transformed into one with four terms and the corresponding four queries are sent to the data source. Clearly, the two-query plan above is preferable, although the same amount of data is transferred in both cases. A CNF system converts the query to one with six clauses, only two of which, $(style = \text{“}sedan\text{”})$ and $(size = \text{“}compact\text{”} \vee size = \text{“}midsize\text{”})$, can be processed directly by the data source. The corresponding query plan involves the transfer of many more entries from the data source than necessary.

Rather than blindly working in DNF or CNF, we believe that a query-processing system that deals with Web data sources must carefully consider the space of available options, and select an efficient one that is indeed supported by the source. □

The examples illustrate that even the seemingly simple task of processing selection queries can be complex when dealing with Web data sources having limited capabilities. In this chapter, we focus our attention on developing techniques for efficiently generating capability-sensitive plans for such selection queries.[1] The reasons for our focus on selection queries are: selection queries are very common in the Web context; and the payoff for identifying good plans for them is very significant, given the slow response time of many Web sources and the low bandwidth of many connections. Also, selection queries form the basis for more complex queries and optimizing their processing is essential to handling the complex queries.

The problem we face may be looked at as a "traditional" query-optimization problem: We need to explore different plans for a given selection query, and choose the one that performs the "best." However, we face some special challenges in this process. The query capabilities of Web sources vary quite a bit. Some sources process only conjunctive query conditions, while some others process disjunctive conditions in a limited way based on Web forms. We have to deal with not only restrictions on the size and structure of condition expressions but also limitations on the sets of attributes returned (see Section 2.2.3). The capricious nature of source restrictions makes it very difficult to generate plans that are a-priori known to be feasible. Instead, a huge number of plans must be explored to ensure that we find the one that is feasible and efficient.

Existing optimizers adopt ad hoc strategies to limit the space of plans considered, consequently making them unable to find good feasible plans in many situations (see the examples above and Section 2.7). Because the payoff for finding efficient plans in the Web context is very high, we focus our attention on exploring large spaces of plans for selection queries. We present a scheme that efficiently explores the large spaces of plans by employing special structures and techniques for compactly representing groups of "related" plans, and for determining their feasibility efficiently. We also develop pruning rules specifically targeted for selection queries in the Web context that help significantly in efficient plan generation.

[1]The material presented in this chapter is based on ideas developed by the author with Wilburt Labio and Hector Garcia-Molina. Some of these ideas had been presented in the conference paper [20].

In Section 2.2, we define a framework for describing source capabilities and query plans. We present our scheme for generating efficient feasible plans for selection queries in Section 2.4. Section 2.3 introduces basic concepts used in our scheme. In Section 2.5, we study the performance of our scheme through analysis and experiments.

2.2 Preliminaries

In this section, we introduce our notation for query plans. The query plans respect the capabilities of the data source as specified using a source-description language discussed in Section 2.2.3.

2.2.1 Notation

Much of the discussion in this chapter focuses on the generation of efficient feasible query plans for a given target selection query q of the form $\pi_A(\sigma_C(R))$, where R is a source relation, C is a Boolean condition expression and A is a set of attributes of R. We call R a "source" as well as a "relation."

The condition expression C of the target query is represented by a *condition tree* (CT for short). Figure 2.1 shows an example of a CT. The leaf nodes of a CT represent Boolean conditions, called *atomic conditions*, that do not have $\wedge$'s or $\vee$'s. The non-leaf nodes of a CT represent the Boolean connectors $\wedge$ or $\vee$. (Labels "n_0", "n_1" and "n_2" are used for later reference.) We assume that negation operations in the condition expression can be pushed all the way to the level of atomic conditions. Hence, we do not consider the negation operator explicitly in condition trees.

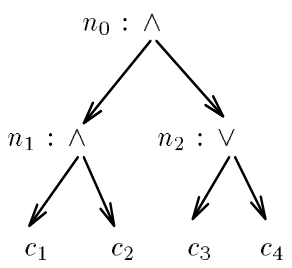

Figure 2.1: Condition Tree.

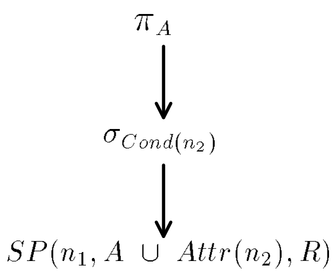

Figure 2.2: Query Tree.

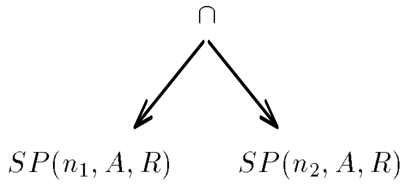

Figure 2.3: Query Tree.

Atomic conditions are denoted c or c_i; condition expressions, possibly involving $\wedge$'s or $\vee$'s, are denoted C or C_i. The condition expression represented by the subtree rooted at some node n of a CT is denoted by $Cond(n)$. For instance, in Figure 2.1, $Cond(n_1)$ is $(c_1 \wedge c_2)$. Given a condition expression C, we denote the set of attributes appearing in C as $Attr(C)$. $Attr(n)$ is shorthand for $Attr(Cond(n))$. We use $SP(C, A, R)$ as an alternative denotation for $\pi_A(\sigma_C(R))$; $SP(C', A', SP(C, A, R))$ denotes $\pi_{A'}(\sigma_{C'}(\pi_A(\sigma_C(R))))$. In the case of a node n of some CT, $SP(n, A, R)$ is shorthand for $SP(Cond(n), A, R)$.

2.2.2 Query Plans

Unlike conventional relational sources, source R may support only a restricted set of SP queries (as described by the source's capability description, see below). Since target queries may not be supported by the source, target queries are submitted to a *mediator* that generates and executes source-capability-sensitive query plans. A mediator's query plan can be represented by a *query tree*. The leaf nodes represent SP queries, called *source queries*, that are actually sent to R. The non-leaf nodes represent selection (σ), projection (π), intersection ($\cap$) and union ($\cup$) operations that are performed at the mediator to combine the results of the source queries.[2] The non-leaf nodes can also represent a *Choice* operator as introduced in [27]. A *Choice* operator is used to concisely represent a set of alternative query plans; each child of the operator represents an alternative query plan. A cost-based module makes the final decision as to which plan to use.

Figures 2.2 and 2.3 show examples of query trees representing plans for the target query $SP(((c_1 \wedge c_2) \wedge (c_3 \vee c_4)), A, R)$. Note that n_0, n_1 and n_2 are nodes of the CT shown in Figure 2.1. $SP(n_2, A, SP(n_1, A \cup Attr(n_2), R))$ is the query plan represented by the query tree in Figure 2.2 and $SP(n_1, A, R) \cap SP(n_2, A, R)$ is the plan of Figure 2.3.

It may be the case that only one of the query plans shown in Figures 2.2 and 2.3 is feasible. For instance, if R cannot answer $SP(n_2, A, R)$, then the second plan is not feasible. Also notice that we do not annotate plan operators as to where they are executed: it is always the case that leaf nodes (operating on R directly) are executed at the source, while the rest are executed at the mediator. For instance, in the first query plan, the mediator performs selection and projection operations on the results of a single source query while in the second plan two source queries are involved. In our SP query notation, whenever the

[2]The mediator may also perform other operations like duplicate elimination, if necessary.

third argument is the source relation, the SP query denotes a source query. Otherwise, the selection and projection operations are carried out at the mediator.

2.2.3 Describing Source Capabilities

Before we discuss how feasible query plans can be generated for a target query, we need a language to describe source capabilities. Using the source-capability description, we can check whether a source query is indeed supported by the source. A plan for the target query is feasible if and only if all the source queries it uses are supported by the source.

Web sources have a wide range of query limitations. Some of these are:

- *Condition Attribute Restrictions*: Some sources do not allow condition specification on certain attributes (e.g., cannot specify a condition on the price attribute of the Barnes and Noble Web bookstore [90]); Some sources require that a particular set of fields be filled in (e.g., must specify stock symbol of a company when querying CheckFree Investment Services stock-quote server [95]).
- *Limiting the Size of the Condition Expression*: Some sources limit the number of conditions in the selection condition expression (e.g., one can specify at most three conditions on the advanced search page of the Library of Congress [93]).
- *Restrictions on the Structure of the Condition Expression*: Some sources allow only atomic condition expressions (e.g., most public library Web pages [94]); Some sources allow only conjunctive queries (e.g., the Junglee Internet shopping guide interfaces [92]); Some sources restrict search expressions based on the structure of a form (e.g., bookstores like [85] and automobile sales sources like [89]).

In addition to these restrictions, it is possible that some sources may allow certain attributes to be projected only when appropriate input attributes are specified. For example, a bank may allow the retrieval of some attributes of an account given its account number, but may refuse to give the account balance unless a PIN number is specified in the query condition.

To describe the selection-query capabilities such as the above, we use a source-capability description language based on context free grammars (CFGs). While the description language we use is quite powerful in representing a variety of query-capability limitations, standard parsing technology can be used to check for the supportability of a query against the capability description very efficiently.

EXAMPLE 2.2.1 Suppose we have a simple source $R(make, model, year, color, price)$. The SP query capabilities of R are described by the following source description.

(1) $\_s \rightarrow \_s_1 \mid \_s_2$

(2) $\_s_1 \rightarrow make = \$m \ \wedge \ price < \$p$

(3) $\_s_2 \rightarrow make = \$m \ \wedge color = \$c$

(4) $ATTRIBUTES :: \ \_s_1 : \ \{make, model, year, color\}; \ \_s_2 : \ \{make, model, year\}$

Note that symbols starting with "_" are nonterminals. We use $\$p$ for integer constants, and $\$c$ and $\$m$ for string constants. (The parser built from the CFG can perform the necessary type checking.) In the above source description, the first three rules are standard CFG rules that describe the condition expressions R can evaluate. For example, Rule (2) states that R can evaluate conditions like $(make = \text{"}BMW\text{"} \wedge price < 40000)$ and Rule (3) states that R can evaluate conditions like $(make = \text{"}BMW\text{"} \wedge color = \text{"}red\text{"})$.

The fourth "rule" in our example describes what attributes can be exported by R. It states that a query matching rule (2) (nonterminal $\_s_1$) can fetch any subset of the attribute set $\{make, model, year, color\}$. If the query matches rule (3) (nonterminal $\_s_2$), the retrievable attributes are $\{make, model, year\}$. To keep our attribute rules simple, we assume that in all cases the *starting nonterminal* $\_s$ derives one or more *condition nonterminals* $\_s_j$, and that attribute sets are only associated with these condition nonterminals. Other nonterminals, not directly derived from $\_s$, do not have attribute sets. If the parsing of a query uses condition nonterminal $\_s_j$, then the query may retrieve the attributes that are associated with $\_s_j$. □

We define a function called *Check* that can be used to test if a selection query is supported by a source. Given a condition expression and a source, this function returns the set of attributes that can be exported by the source when evaluating the condition expression. If the condition expression cannot be evaluated by the source, *Check* returns the empty set.[3] Thus, a source query $SP(C, A, R)$ is supported by R if $A \subseteq Check(C)$.

[3]Under certain unusual circumstances, *Check* needs to return more than just the attribute set. In particular, if a source can evaluate a condition, without returning any data for matching tuples (it only tells whether matching tuples exist or not), *Check* needs to return information indicating whether the condition can be evaluated at the source, in addition to identifying the set of exported attributes. We expect such sources to be very rare, and hence work with the simpler *Check* function in the rest of the chapter.

To illustrate the interaction between the source description and feasible plan generation, consider the following target query:

$$SP((make = \text{“}BMW\text{”} \wedge price < 40000) \wedge (color = \text{“}red\text{”} \vee color = \text{“}blue\text{”})), \{model\}, R)$$

Figure 2.1 shows the condition tree, with c_1, c_2, c_3 and c_4 representing the four atomic conditions. Figures 2.2 and 2.3 show two possible query plans for the target query, where $A = \{model\}$. Notice that the plan represented by Figure 2.3 is not feasible because $SP(n_2, A, R)$ is not supported by R, i.e., $Check(Cond(n_2))$ returns the empty set. On the other hand, $Check(Cond(n_1))$ returns $\{make, model, year, color\}$, a superset of $A \cup Attr(n_2)$. So, $SP(n_1, A \cup Attr(n_2), R)$ is a supported query. Hence, the plan of Figure 2.2 is feasible.

2.3 A Modular Scheme

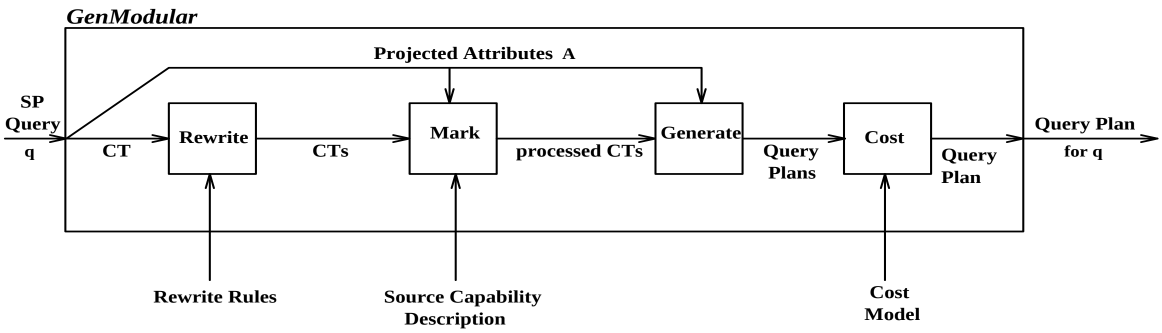

Figure 2.4: *GenModular* Scheme for Generating Query Plans.

In this section, we present a modular scheme, called *GenModular*, for generating efficient feasible query plans for target SP queries. This scheme is a naive, exhaustive one. Our goal in presenting it is to show conceptually how one could go about exploring the set of feasible plans for a given target query. In doing so, we provide a framework that is useful for understanding the much more efficient scheme of Section 2.4. Thus, keep in mind that for *GenModular* the goal is clarity, not efficiency.

GenModular considers various rewritings of the target-query condition, identifies parts of the condition that can be answered by the source, pieces together the source queries for these parts into query plans for the target query, and chooses the least expensive among these plans. As shown in Figure 2.4, four modules work together in *GenModular* to achieve these tasks: *rewrite*, *mark*, *generate* and *cost* (hence the "modular" in the name). In this section, we describe the rewrite, mark and generate modules in detail. The cost module is not described in detail since its operation is standard. (It selects the best plan from a set of plans, using whatever cost model is applicable.)

2.3.1 Rewrite Module

The rewrite module produces a set of equivalent rewritings of the target-query condition. It uses a set of rules that are also input to the module. Examples of useful rewrite rules include commutative, associative and distributive transformations of condition expressions. Rules like $C \equiv C \vee C$ and $C \equiv C \wedge C$ (we call them *copy* rules) are also quite useful. In our experimental studies on a set of Web sources, we observed that using these rules we could generate plans that are far superior to those obtained by contemporary systems (see Section 2.5).

The target-query condition as well as its rewritings are represented by condition trees (CTs). The rewrite module produces a set of CTs that represent condition expressions equivalent to the target-query condition.[4] These CTs are passed on to the mark module.

2.3.2 Mark Module

For each CT produced by the rewrite module, the mark module determines the various parts of the CT that can be evaluated at the source. To make this determination, the *Check* function (defined in Section 2.2) is called on the various subtrees of the CT. If a part of the CT can be evaluated at the source, the mark module indicates the set of attributes returned by the source when evaluating that part. The following example illustrates how CTs are marked in this module.

[4]Standard techniques to ensure termination of the rewriting process are employed here. That is, as rewrite rules are applied, the rewrite module needs to visit the various nodes of a transformation graph (each node representing a different rewriting of the target condition) and avoid rewriting the same condition tree with the same rewrite rule repeatedly.

EXAMPLE 2.3.1 Consider the source description of R given in Example 2.2.1 and the following target query:

$$SP((price < 40000 \wedge color = \text{"}red\text{"} \wedge make = \text{"}BMW\text{"}), \{model, year\}, R).$$

Let t_1 be the CT for the target-query condition. The rewrite module produces among others the CT, say t_2, representing:

$$((make = \text{"}BMW\text{"} \wedge price < 40000) \;\wedge\; (make = \text{"}BMW\text{"} \wedge color = \text{"}red\text{"}))$$

When processing t_2, the mark module invokes $Check(Cond(n_0))$ where n_0 is the root node of t_2. R cannot evaluate $Cond(n_0)$, so this call to $Check$ returns the empty set. Each node n in the CT has a field $n.export$ that records the set of attributes that can be exported by R when asked to evaluate $Cond(n)$. The mark module uses the $Check$ function to compute the $export$ fields of all the nodes in the CT. In our example, $n_0.export$ is computed to be the empty set.

Notice that the mark module must process *every* CT node even if one of its ancestors represents a condition that can be evaluated at R. Otherwise, we may miss the possibility of evaluating some part of the CT at R and evaluating the rest of the CT at the mediator.

Continuing with our example, let the first child of n_0 be n_1 (representing the condition $(make = \text{"}BMW\text{"} \wedge price < 40000)$). $Cond(n_1)$ can be evaluated at R and the exported set of attributes $\{make, model, year, color\}$ is stored in $n_1.export$. Similarly, for the other child of n_0, denoted n_2, the mark module computes $n_2.export$ as $\{make, model, year\}$. All other nodes of t_2 will end up with empty $export$ fields, because none of the condition expressions represented by these nodes can be evaluated at R. For the same reason, the $export$ fields of all the nodes of t_1 are computed to be empty, denoting that no part of the CT t_1 can be evaluated at R. □

2.3.3 Generate Module

In this section we present an algorithm called *EPG* (for Exhaustive Plan Generator) that computes the feasible plans for a given CT. The generate module produces the set of feasible plans by repeatedly invoking *EPG* on each of the CTs passed on by the mark module. The following example illustrates how *EPG* works.

EXAMPLE 2.3.2 Consider the two CTs, t_1 and t_2 of Example 2.3.1. *EPG* performs a pre-order traversal of each CT. Focusing on t_2, when *EPG* processes n_0, it checks if $n_0.export$ contains the requested attributes of the target query, denoted A. If so, it generates the plan $SP(n_0, A, R)$. We call this type of plan a *pure* plan for n_0 because $Cond(n_0)$ is evaluated in its entirety at the source.

However, $n_0.export$ is empty in our example, so the pure plan for n_0 is not feasible. Regardless of whether this plan is feasible or not, *EPG* proceeds to examine the children of n_0 to find other plans, because it aims to generate all the feasible plans from the CT. Other plans can be generated for n_0 by recursively computing plans for each of its children nodes (i.e., n_1 and n_2) and combining them together. For instance, this strategy yields the following feasible plan for n_0: $SP(n_1, A, R) \cap SP(n_2, A, R)$. We call such a plan *impure* for n_0 because $Cond(n_0)$ is evaluated piecemeal.

There can be other feasible impure plans for n_0. For instance, source R supports the query $SP(n_1, \{make, model, year, color\}, R)$. Then $Cond(n_2)$ can be locally evaluated at the mediator based on the result of this source query, instead of remotely at the source. Thus, we have the following feasible plan: $SP(n_2, A, SP(n_1, A \cup Attr(n_2), R))$. In general, for each node like n_0, *EPG* generates plans where a subset of n_0's children are evaluated at the source, and the rest are evaluated locally at the mediator.

EPG generates the above two feasible plans based on t_2. No feasible plans can be generated from t_1 because all the nodes in t_1 have empty *export* fields indicating that no part of that CT can be evaluated at R. □

The previous example illustrates how *EPG* generates query plans given a CT. We now discuss the formal algorithm for *EPG* (Figure 2.5). Each call to *EPG* is a request to generate a set of plans for a query $SP(n, A, R)$, where n is a node of a CT. *EPG* uses the *Choice* operator to concisely represent the set of plans it generates for n.

As illustrated in Example 2.3.2, *EPG* first checks (Line 2) if $SP(n, A, R)$ is supported by R. If so, this pure plan is generated in Line 3. To find the impure plans for an $\wedge$ node n, *EPG* generates plans for each of n's children nodes and combines them to generate plans for n (Line 5). In addition, in Lines 7–8, it generates plans for a subset of n's children and evaluates the rest of the children locally. In Figure 2.5, *Local* denotes the unevaluated children nodes (Line 7) and $AND(Local)$ denotes the unevaluated condition (i.e., $AND(Local)$ represents the conjunction of each $Cond(n')$ where n' is in *Local*). Notice that the recursive *EPG* call in Line 8 must request additional attributes required to evaluate

Algorithm 2.3.1 Algorithm *EPG*
Input: n, A, R
Output: The set of feasible query plans for $SP(n, A, R)$
Method:

1. $PLANS \leftarrow \{\}$
2. If $(A \subseteq n.export)$
    3. $PLANS \leftarrow PLANS \ \cup \ \{SP(n, A, R)\}$
4. If n is an $\wedge$ node
    5. $PLANS \leftarrow PLANS \ \cup \ \{\bigcap_{n' \in n.children} \ EPG(n', A, R)\}$
    6. For each non-empty proper subset $Remote$ of $n.children$
        7. $Local \leftarrow n.children - Remote$
        8. $PLANS \leftarrow PLANS \ \cup$
        $\{SP(AND(Local), A,$
        $\bigcap_{n' \in Remote} EPG(n', A \cup Attr(AND(Local)), R))\}$
9. Else If n is an $\vee$ node
    10. $PLANS \leftarrow PLANS \ \cup \ \{\bigcup_{n' \in n.children} \ EPG(n', A, R)\}$
11. If $((A \cup Attr(n)) \subseteq Check(\texttt{true}))$
    12. $PLANS \leftarrow PLANS \cup \{SP(n, A, SP(\texttt{true}, A \cup Attr(n), R)) \ \}$
13. If $PLANS = \{\}$
    14. Return(ϕ)
15. Else
    16. Return$(Choice(PLANS))$ $\diamond$

Figure 2.5: Algorithm EPG.

$AND(Local)$.

If n is an $\vee$ node (Lines 9–10), *EPG* again generates a plan for n by combining the plans for all the children of n. Note that there is no opportunity to generate plans that evaluate parts of a disjunction using source queries and evaluate the other parts of the disjunction on the results of these source queries.

Finally, *EPG* explores the possibility of downloading the relevant portion of the source contents and evaluating the entire $Cond(n)$ at the mediator. The corresponding plan would send a source query with a trivially-true condition, projecting out the requested attributes A along with the attributes necessary to evaluate n (Lines 11–12).

If *EPG* finds no plans for n, it returns ϕ to indicate that $SP(n, A, R)$ cannot be evaluated in any way. Any query plan that makes use of ϕ (for instance, when combining the plans from recursive calls to *EPG* in Lines 5, 8 and 10) is automatically eliminated from the set of plans output by the generate module. (This check for empty plans could be done within procedure *EPG*, but we do not show it for simplicity.) If *EPG* finds feasible plans for n, it concisely represents the plans using the *Choice* operator (Line 16) that indicates the alternative query plans for $SP(n, A, R)$.

2.4 GenCompact

In the previous section, we discussed *GenModular*, a scheme to generate efficient feasible plans for target queries. One drawback of *GenModular* is that it is very inefficient as it exhaustively enumerates all the possible plans and then selects the one that is most efficient. In this section, we present a scheme, called *GenCompact*, that can generate the same plans in a much more efficient manner. *GenCompact* improves upon *GenModular* as follows:

- Intelligent Plan Generation: By using a more intelligent plan-generation module, that integrates the mark, generate and cost modules, and implicitly handling some of the rewrite rules, *GenCompact* significantly reduces the number of CTs that need to be processed.
- Pruning Techniques: By using knowledge of the cost model, it identifies strategies to significantly reduce the plan space explored *without* pruning the optimal plan.

The reader should keep in mind that, although *GenCompact* is much more efficient than *GenModular*, it may still be computationally expensive, since it must consider many options to find a good plan. The performance characteristics of *GenCompact* are discussed in Section 2.5.

We begin our discussion of *GenCompact* by describing the simplified rewrite module in Section 2.4.1. Section 2.4.2 presents our cost model upon which the pruning rules of Section 2.4.3 are based. Finally, in Section 2.4.4, we present the plan-generation module of *GenCompact*.

2.4.1 Rewrite Module

As in the *GenModular* scheme, *GenCompact* employs a rewrite module to construct a set of CTs equivalent to the CT representing the target-query condition.[5] However, *GenCompact*

[5] As in the case of *GenModular*, the termination of the rewrite process is achieved by techniques that traverse the transformation graph such that a condition tree is not rewritten in the same way more than

can work with a lot fewer CTs than *GenModular*. In particular, *GenCompact*'s rewrite module fires fewer rewrite rules, without compromising the optimality of the plans being generated. The commutativity rule is eliminated by rewriting the source description as discussed later in this section. In addition, the associativity and the copy rules are dropped because of the implicit handling of these rules in *GenCompact*'s improved plan-generation module (described in Section 2.4.4). By reducing the set of rewrite rules fired and producing fewer CTs, much of the redundant work performed by *GenModular* is avoided. For instance, in *GenModular*, two CTs that differ by one application of a rewrite rule are processed independently leading to redundant work.

The elimination of the commutativity, associativity and copy rules from the rewrite module is a "safe" optimization in the sense that it provides significant performance improvements to *GenCompact* without impacting its ability to generate efficient feasible plans for target queries. Although the commutativity, associativity and copy rules can be eliminated, other rules may still need to be used to find the optimal plan. An obvious opportunity for unsafe pruning is to drop some of these rules or restrict their use so that fewer CTs are produced. For instance, one may choose to limit the use of the distributivity rule so that it produces only the CNF and DNF versions of the target query CT, and not use any other rule. Consequently, *GenCompact* can work with just three CTs – the original CT, the CNF CT and the DNF CT. An extreme case of unsafe optimization, but one that may turn out to be quite practical, is to drop the rewrite module altogether and have *GenCompact* process a single CT, the one produced directly from the target-query condition.

Rewriting Source Capability Description

To illustrate how we handle commutativity, consider once again Example 2.2.1, with the following rule:

$$(3)\ \_s_1 \quad \rightarrow \quad make = \$m \wedge color = \$c.$$

This rule specifies that the condition on *make* must come before the condition on *color*. If the target-query condition is ($color =$ "*red*" $\wedge\, make =$ "*BMW*"), it cannot be evaluated at the source. (Of course, for many sources order is not important. However, for this example it is, because that is what the grammar specifies.) In this case, the commutativity rule

once.

helped *GenModular* identify the equivalent condition ($make$ = "*BMW*" $\wedge$ $color$ = "*red*") to generate a feasible plan for the target query. Instead of firing the commutativity rule, one can rewrite R's source description to make R *appear* order insensitive. For instance, given the rule above, we can add the rule

$$(3')\ \_s_1 \quad \rightarrow \quad color = \$c \wedge make = \$m.$$

Of course, when the mediator actually sends a source query, it must "fix" it so the correct order is respected. The overhead incurred in fixing the source queries is low since the mediator only fixes the source queries of the one query plan that is to be executed and not of every possible query plan considered. On the other hand, when the source description is not rewritten and the commutativity rule is used, a much larger set of CTs needs to be processed at plan-generation time.

Note that some of the work that used to be performed by the rewrite module of *GenModular* is now going into rewriting the CFG source description, and parsing with a larger set of rules. However, there is an important difference. Source-description rewriting is undertaken only once when the source is integrated into the system, not every time a target query is processed. Moreover, by increasing the number of CFG rules, we only increase the complexity of building the parser, which is done not at run time, but when the source joins the system. When verifying whether a source query is supported, the parser still runs in time linear in the size of the selection condition and not the number of CFG rules in the source description.

2.4.2 Cost Model

GenModular does not take advantage of the cost model and instead exhaustively generates all the possible plans and then evaluates their costs. In contrast, *GenCompact* takes advantage of the cost model by pruning its search space of query plans. The set of pruning rules used for this purpose are dependent on the cost model.

We use a cost model that we believe is well suited for Web query processing. In this model, given a query plan that uses a set of source queries SQ, the cost of the plan is

$$\sum_{sq\ \in\ SQ} k_1 + k_2(result\ size\ of\ sq). \tag{2.1}$$

Constant k_1 represents the overhead in sending the messages over the Web and the overhead in starting a query at source R (e.g., delay in connecting to the Web server). Constant k_2 represents the cost per tuple of computing the answer at source R and the cost per byte of transferring the answer over the network. Note that k_1 and k_2 depend on the source. That is, different sources can have different k_1 and k_2 values in our cost model.

Although mediator costs are not explicitly taken into account in the cost model, minimizing the cost based on Equation (2.1) will likely lower mediator cost as well. Minimizing Equation (2.1) involves reducing the number of source queries and the amount of data transferred into the mediator. Correspondingly, fewer mediator operations may be required since there are fewer source queries whose results need to be combined. Furthermore, when the results of the source queries are combined, less data needs to be processed.

2.4.3 Pruning Rules

Based on our cost model, we formulate the following pruning rules:

pr_1: *Prune impure plans when pure plan exists.* A pure plan processes the target query entirely at the source (no mediator postprocessing is required). If the pure plan is feasible, no impure plan need be generated, because under our cost model it will never be cheaper than the pure plan. The pure plan is preferred because impure plans use at least as many source queries and transfer at least as much data as the pure plan.

pr_2: *Prune locally-suboptimal plans.* In order to find impure plans for a target query $SP(n, A, R)$, we break it up into many subqueries, generate plans for the subqueries and combine them to form the target-query plans. When considering plans to be combined, in our cost model it is safe to prune away all but the cheapest plan for each subquery.

pr_3: *Prune dominated plans.* This rule applies only to queries with conjunctive conditions (see Section 2.4.4). Suppose, the target query is $SP((c_1 \wedge c_2 \wedge c_3 \wedge c_4), A, R)$. Let P_1 denote a plan for the subquery $SP(c_1 \wedge c_2 \wedge c_3, A, R)$ and P_2 denote a plan for the subquery $SP(c_1 \wedge c_2, A, R)$. If P_1 is cheaper than P_2, we say P_1 dominates P_2. In such a case, there is no need to consider P_2 when combining plans to form the target-query plan. We can prune away P_2 because the cost of any plan P for the target query that uses P_2 can always be lowered by replacing P_2 with P_1 in P.

These three pruning rules are extensively used in the plan-generation module of *GenCompact* and they yield rich dividends as evidenced by the performance of *GenCompact* (see Section 2.5).

2.4.4 Plan Generation Module

The plan-generation module takes each CT produced by the rewrite module and generates a single query plan (without *Choice* operators) for the CT. As indicated earlier, the associativity and copy rules are not used in the rewrite module. Instead, the plan-generation module implicitly handles these rules by doing more work on each CT it receives from the rewrite module. In particular, for each of its input CTs, it not only considers all the plans that accrue directly from the CT but also explore all the plans that accrue from the CTs that can be obtained by applying the associativity and copy rules on the CT. To facilitate the exploration of all these plans, the plan-generation module starts by converting each CT it receives from the rewrite module into its equivalent *canonical* CT. A CT is in canonical form if the children of every $\wedge$ node are either leaf or $\vee$ nodes and the children of every $\vee$ node are either leaf or $\wedge$ nodes. For instance, the CT representing ($price < 40000 \wedge color =$ "red" $\wedge$ $make =$ "BMW") is canonical because all of the root $\wedge$ node's three children are leaf nodes. On the other hand, the CT representing ($price < 40000 \wedge (color =$ "red" $\wedge$ $make =$ "BMW")) is not canonical since the root $\wedge$ node has two children, one of which is an $\wedge$ node. Converting a CT into an equivalent canonical tree can be done in time linear in the size of the input CT.

2.4.5 Integrated Plan Generator

For each CT received from the rewrite module, the plan-generation module invokes *IPG* (Integrated Plan Generator) on the corresponding canonical CT, to obtain the best plan for the target query based on that CT. After obtaining the best plan for each CT, the overall best plan is chosen.

IPG (see Figure 2.6) first checks if the pure plan $SP(n, A, R)$ is feasible. If so, using pruning rule pr_1, *IPG* avoids any further search and returns the pure plan for the target query. Otherwise, *IPG* tries to find the cheapest feasible impure plan. One impure plan to consider is to download the relevant portions of the source and evaluate $Cond(n)$ at the mediator. It stores this plan in $plan_{impure}$. If it is not feasible to download the source

Algorithm 2.4.1 Algorithm *IPG*
Input: n, A, R
Output $plan_{pure}$ or $plan_{impure}$ // the best plan for $SP(n, A, R)$
Method:
If $A \subseteq Check(Cond(n))$
  $plan_{pure} = SP(n, A, R)$ // the pure plan
  Return($plan_{pure}$)
If $A \cup Attr(n) \subseteq Check(\texttt{true})$
  $plan_{impure} = SP(n, A, SP(\texttt{true}, A \cup Attr(n), R))$
Else // downloading R is not feasible
  $plan_{impure} = \phi$
If n is a leaf node
  Return($plan_{impure}$)
Else If n is an $\vee$ node
  Execute code in Figure 2.7
Else If n is an $\wedge$ node
  Execute code in Figure 2.8 $\diamond$

Figure 2.6: Algorithm IPG.

contents, $plan_{impure}$ will be ϕ, indicating that it is infeasible. Now, if n is a leaf node, there are no other impure plans to be considered. However, if n is not a leaf node, there are opportunities to explore other impure plans. We discuss how these plans are explored and complete the *IPG* algorithm below.

Notice that *IPG*, like the *EPG* algorithm of *GenModular* (Section 2.3.3), traverses the CT in pre-order. Unlike *EPG*, however, *IPG*'s CT traversal can stop without processing all the nodes of the CT, based on pruning rule pr_1.

2.4.6 Processing an $\vee$ Node

When *IPG* processes an $\vee$ node and the pure plan is not feasible, it looks for feasible impure plans. The following example illustrates this process.

EXAMPLE 2.4.1 Consider a target query $SP(c_1 \vee c_2 \vee c_3 \vee c_4, A, R)$. Suppose that the following four queries are supported by R: $SP(c_1 \vee c_2, A, R)$, $SP(c_1 \vee c_2 \vee c_3, A, R)$, $SP(c_2 \vee c_3 \vee c_4, A, R)$, $SP(c_1 \vee c_4, B, R)$ (with $B \subset A$).

Since R does not support the condition expression of the target query, there is no feasible pure plan. However, there are a number of feasible impure plans for the target query. The cheapest among these, under our cost model, turns out to be

$$SP(c_1 \vee c_2, A, R) \;\cup\; SP(c_2 \vee c_3 \vee c_4, A, R). \tag{2.2}$$

We now show how the various query plans are explored and how Plan (2.2) is chosen.

Let the canonical CT representing the target-query condition be t_0. Having determined that the pure plan for the target query is infeasible, *IPG* finds the feasible subplans for parts of the disjunctive condition. More precisely, when an $\vee$ node n is processed, *IPG* checks if the disjunction of each non-empty subset of n's children can be evaluated at R. When the root node of t_0 is processed, *IPG* determines that the only four subsets that can be evaluated by R are: $\{c_1, c_2\}$, $\{c_1, c_2, c_3\}$, $\{c_2, c_3, c_4\}$ and $\{c_1, c_4\}$. The corresponding four subplans are: $SP((c_1 \vee c_2), A, R)$, $SP((c_1 \vee c_2 \vee c_3), A, R)$, $SP((c_2 \vee c_3 \vee c_4), A, R)$, and $SP((c_1 \vee c_4), B, R)$. The fourth subplan is not useful because it does not project enough attributes.

Once the three useful subplans are identified, *IPG* constructs query plans for the root node of t_0 by combining the subplans using a mediator union (since an $\vee$ node is being processed). Of course, each combination of the subplans must "cover" all the children of the root node. Only three combinations of subplans qualify:

1. $SP((c_1 \vee c_2), A, R) \cup SP((c_2 \vee c_3 \vee c_4), A, R)$
2. $SP((c_1 \vee c_2 \vee c_3), A, R) \cup SP((c_2 \vee c_3 \vee c_4), A, R)$
3. $SP((c_1 \vee c_2), A, R) \cup SP((c_1 \vee c_2 \vee c_3), A, R) \cup SP((c_2 \vee c_3 \vee c_4), A, R)$

Notice that the first plan, identical to Plan (2.2), is clearly the best plan. The second plan is more expensive than Plan (2.2) since more data is transferred. The third plan is even worse since more data is transferred and more source queries are used. Thus, Plan (2.2) is chosen. $\square$

As illustrated in the example, when the pure plan is not feasible, there are two steps in finding the best feasible impure plan for an $\vee$ node, n. First, feasible subplans that evaluate "parts" of $Cond(n)$ are identified. Second, the best query plan that combines a subset of the subplans is chosen. Figure 2.7 describes the process more precisely.

Algorithm 2.4.2 $\wedge$ Case of Algorithm *IPG*
Input: n, A, R
Output: $plan_{impure}$ // the best plan for $SP(n, A, R)$
Method:

// **Step 1**: Find subplans
1. For each non-empty subset N of $n.children$
   2. $P[N] \leftarrow \phi$
3. For each non-empty subset N of $n.children$
   4. If $A \subseteq Check(OR(N))$
      5. $P[N] \leftarrow SP(OR(N), A, R)$ // the pure plan for $OR(N)$
6. For each *child* n' of n such that $P[\{n'\}] = \phi$
   7. $P[\{n'\}] \leftarrow IPG(n', A, R)$ // impure plan for $OR(\{n'\})$

// **Step 2**: Solve the MCSC problem and combine the subplans
8. $E \leftarrow$ all subsets N of $n.children$ such that $P[N] \neq \phi$
9. For each subset SC of E that evaluates all of n's children
   10. $plan \leftarrow \bigcup_{N \in SC} P[N]$
   11. If $Cost(plan) < Cost(plan_{impure})$
       12. $plan_{impure} \leftarrow plan$
13. Return($plan_{impure}$) // the best impure plan for n $\diamond$

Figure 2.7: $\vee$ Node Processing of Algorithm IPG.

Given a query $SP(n, A, R)$, Lines 1–5 identify the subplans for parts of $Cond(n)$ that project at least A and store them in a subplan array P. This array has an entry for each possible subset of n's children. For example, $P[\{n_1, n_2\}]$ stores the best known plan for $Cond(n_1) \vee Cond(n_2)$, where n_1 and n_2 are children of n. In Figure 2.7, $OR(N)$ denotes the disjunction of each $Cond(n')$ where n' is in N. As usual, ϕ denotes an invalid plan.

Other subplans are obtained, if necessary, by recursively calling *IPG* on each child n' of n (Lines 6–7). Here, we are looking for subplans that are not generated by Lines 1–5 in the case of singleton N. If by Line 6 $P[\{n'\}]$ has a valid plan (not ϕ), a pure plan has been found for n', and using pruning rule pr_1, we can avoid searching for impure plans for n'. Otherwise, *IPG* computes the best impure plan for $SP(n', A, R)$ in Line 7. Because of pruning rule pr_2, we can ignore the other subplans for n' and keep track of just the one returned by the recursive call to *IPG*. Note that when $P[\{n'\}]$ is invalid at Line 6, *IPG* still needs to be called recursively even when there is a valid plan $P[N]$ that evaluates more nodes (i.e., $\{n'\} \subset N$). While $P[N]$ evaluates more nodes than the subplan that results from the recursive *IPG* call, it may cost more because of increased amount of data transfer. Thus, plans using $P[\{n'\}]$ may be better than plans using $P[N]$ when N is a strict superset of $\{n'\}$.

Once the subplans are found, the second step (Lines 8–13) chooses a set of subplans with minimum total cost that together evaluate the entire $Cond(n)$. This set of subplans are combined using a mediator union (Line 10). The problem of choosing such a set of subplans is the well known "Minimum Cost Set Cover" (MCSC) problem [31]. Since MCSC is known to be $\mathcal{NP}$-complete, to obtain the optimal set of subplans, *IPG* examines all possible sets of subplans. Even though *IPG* would have worst-case time complexity that is exponential in the number of subplans it constructs, as we discuss in Section 2.5, in many scenarios *IPG* performs very well as the number of subplans over which it solves the MCSC problem turns out to be small.

Note that, by considering the various sets of subplans in Line 9, *IPG* accounts for all the plans that would be considered by *GenModular* from equivalent CTs obtained using the associativity and copy rewrite rules. Hence, *GenCompact* is able to drop those rules from its rewrite module without compromising the quality of the target-query plans it finds.

2.4.7 Processing an $\wedge$ Node

The plan-generation process to find the best impure plan for an $\wedge$ node is similar to that of an $\vee$ node. That is, the processing is again divided into two steps: (1) find feasible subplans for sets of children nodes; and (2) choose the best combination of subplans. However, there are significant differences in how the first step is performed, as we illustrate in the following example.

EXAMPLE 2.4.2 Consider the target query $SP((c_1 \wedge c_2 \wedge c_3), A, R)$. Suppose that R supports the following three queries:

1. $SP(c_1, A, R)$
2. $SP(c_2, A \cup Attr(c_3), R)$
3. $SP(c_3, A \cup Attr(c_2), R)$

The pure plan for the target query is not feasible. However, there are three feasible impure plans for the target query based on the source queries supported by R.

$$SP(c_1, A, R) \cap SP(c_2, A, R) \cap SP(c_3, A, R) \tag{2.3}$$

$$SP(c_1, A, R) \cap SP(c_3, A, SP(c_2, A \cup Attr(c_3), R)) \tag{2.4}$$

$$SP(c_1, A, R) \cap SP(c_2, A, SP(c_3, A \cup Attr(c_2), R)) \tag{2.5}$$

We now illustrate how these plans are explored in *IPG*.

The first step in producing the plans for an $\wedge$ node, denoted n, is to find subplans that evaluate parts of $Cond(n)$. Just like in the case of an $\vee$ node, *IPG* considers each non-empty subset of n's children, denoted N, and checks if $AND(N)$ can be evaluated at R. In this example case, the only subsets that qualify are: $\{c_1\}$, $\{c_2\}$ and $\{c_3\}$.

Unlike in the case of $\vee$ node processing, we have opportunities to compute feasible subplans for other subsets of n. For instance, c_3 can be locally evaluated on the result of the source query $SP(c_2, A \cup Attr(c_3), R)$ that covers the children set $\{c_2\}$. This subplan is used in Plan (2.4).

As in the case of the $\vee$ node processing, the subplans for the various subsets of the children of n are stored in the subplan array P. Notice that, unlike in the $\vee$ node case, for a given subset of children of n, there may exist multiple subplans. Using pruning rule pr_2, we keep the best among these and drop others from further consideration. Fortunately, we can prune away all but the cheapest subplan for a given subset of children. This kind of pruning is safe because we can make a local cost-based decision that is guaranteed to not lose the optimal plan. Let the subplan $SP(c_3, A, SP(c_2, A \cup Attr(c_3), R))$ be cheaper than the subplan $SP(c_2, A, SP(c_3, A \cup Attr(c_2), R))$. Then the entries of P are: (1) $P[\{c_1\}] = SP(c_1, A, R)$; (2) $P[\{c_2\}] = SP(c_2, A, R)$; (3) $P[\{c_3\}] = SP(c_3, A, R)$; and (4) $P[\{c_2, c_3\}] = SP(c_3, A, SP(c_2, A \cup Attr(c_3), R))$. Given the supported queries we assumed, no other feasible subplans exist.

Once all the feasible subplans are found, the second step of *IPG* chooses a set of subplans that evaluates all the children nodes of n with the minimum cost (again the MCSC problem). There are two candidate sets of subplans to consider: $\{P[\{c_1\}], P[\{c_2\}], P[\{c_3\}]\}$ and $\{P[\{c_1\}], P[\{c_2, c_3\}]\}$. The first set yields Plan (2.3) and the second set yields Plan (2.4). Notice that Plan (2.5) is not even considered since it is guaranteed to be no more efficient than Plan (2.4). The costs of both Plans (2.3) and (2.4) are evaluated and the cheaper plan is returned. $\square$

Algorithm 2.4.3 $\wedge$ Case of Algorithm *IPG*
Input: n, A, R
Output: $plan_{impure}$ // the best plan for $SP(n, A, R)$
Method:

// **Step 1**: Find subplans
1. For each non-empty subset N of $n.children$
  2. $P[N] \leftarrow \phi$
3. For each non-empty subset N of $n.children$
  4. If $A \subseteq Check(AND(N))$
    5. $P[N] \leftarrow SP(AND(N), A, R)$ // the pure plan evaluating $AND(N)$
    6. $A_N = Check(AND(N))$
    7. $N_{add} \leftarrow MaxEval(A_N, n) \; - \; N$ // see text for *MaxEval* definition
    8. For each non-empty subset N' of N_{add}
      9. $P[N \cup N'] \leftarrow min_{cost}(P[N \cup N'],$
        $SP(AND(N'),\ A,\ SP(AND(N),\ A \cup Attr(AND(N')),\ R)))$
10. For each child n' of n
  11. For each subset N' of $n.children$, containing n'
    12. If there is no N'' such that $N' \subseteq N''$ and $P[N'']$ is a pure plan
      13. $P[N'] \leftarrow min_{cost}(P[N'],\ SP(AND(N' - \{n'\}),\ A,$
        $IPG(n',\ A \cup Attr(AND(N' - \{n'\})),\ R)))$

// **Step 2**: Solve the MCSC problem and combine the subplans
14. Prune all *dominated* subplans from P
15. $E \leftarrow$ all subsets N of $n.children$ such that $P[N] \neq \phi$
16. For each non-empty subset SC of E that evaluates all of n's children
  17. $plan \leftarrow \bigcap_{N \in SC} P[N]$
  18. If $Cost(plan) < Cost(plan_{impure})$
    19. $plan_{impure} \leftarrow plan$
20. Return($plan_{impure}$) // the best impure plan for n $\diamond$

Figure 2.8: $\wedge$ Node Processing of Algorithm IPG.

We now discuss in detail the portion of *IPG* that produces the best impure plan for $SP(n, A, R)$ when n is an $\wedge$ node (Figure 2.8). *IPG* first examines each non-empty subset N of n's children and checks if the query $SP(AND(N), A, R)$ is supported by R. As illustrated in Example 2.4.2, each supported query may produce a set of subplans. This set of subplans is recorded in the subplan array P. Lines 1–2 initialize the subplan array and Lines 3–9 compute feasible subplans.

For a given supported query $SP(AND(N), A, R)$, denoted q_N, the set of feasible subplans based on q_N is computed as follows. The key idea is that the set of attributes A_N that can be exported by R when evaluating $AND(N)$ may be sufficient for the mediator to evaluate other children of n. All such additional children of n are gathered into a set N_{add}. To compute N_{add}, we use a function called $MaxEval$. This function takes the set of attributes A_N and an $\wedge$ node n and returns the set of children of n that can be evaluated using A_N. N_{add} is simply $MaxEval(A_N, n) - N$. Given N_{add}, we can obtain the various sets of children of n that can be evaluated based on the source query q_N. These are computed by Lines 8–9. Notice that when multiple subplans exist for a given set of children, Line 9 uses pruning rule pr_2 to prune away all but the cheapest one.

Lines 10–13 generate other feasible subplans by invoking *IPG* recursively on each child of n. *IPG* looks for opportunities to evaluate sets of children of n, denoted N', based on the plan generated for a particular child of n, denoted n'. Note that in order to evaluate the additional children, the attributes participating in their conditions need to be exported by the plan to evaluate n'. Line 12 uses pruning rules pr_1 and pr_3 to avoid making unnecessary recursive calls to *IPG*. Observe that if $N' = N''$, pr_1 applies and if $N' \subset N''$, pr_3 applies. Line 13 employs pruning rule pr_2 to prune away unnecessary, expensive subplans.

Once all the feasible subplans are determined, the best combination of the subplans that evaluates all the children of n is found by solving the MCSC problem. These subplans are then combined using a mediator intersection to obtain the best impure plan for $SP(n, A, R)$ (Lines 15–20). Before we solve the MCSC problem, we employ pruning rule pr_3 in Line 14 to eliminate as many subplans as possible. To see why there are opportunities for pr_3 in Line 14 note that the application of pr_3 in Line 12 only prunes those impure subplans that are dominated by pure subplans. In Line 14 we prune impure subplans that are dominated by other impure subplans. In addition, we prune dominated pure subplans.

2.5 Performance Evaluation

In this section, we study the performance of *GenCompact*. Specifically, we present results of experiments we conducted that throw light on how its running-time behavior in practical scenarios is and how well it finds good plans. We compare its quality of plans with other standard schemes like CNF and DNF transformations that existing systems employ. For our experiments, we implemented a *GenCompact* version that did not use any rewrite rules.

This version of *GenCompact* worked with a single condition tree for each query that is processed. In particular, since our version of *GenCompact* did not have the distributive rewrite rule, the CNF and DNF schemes could beat it. We deliberately chose such a version of *GenCompact* to see how well its plans measured against those produced by the CNF and DNF schemes.

2.5.1 Efficiency of *GenCompact*

It is obvious that the analytical worst-case running time of *GenCompact* is exponential in terms of the input-query parameters. However, due to the sophisticated pruning techniques it employs, it attempts to cut down on most of the search and exploring tasks. In practice, it may perform quite well in most scenarios.

To understand the practical performance of *GenCompact*, we implemented the *GenCompact* scheme and measured the work performed in processing queries in practice. In one scenario, we examined four sources (the Library of Congress Web site, the Amazon.com Web bookstore, the Junglee.com Web site, and a source that processes conjunctive queries). For each source, we hand-formulated a simple query (with 2 or 3 atomic conditions) and a complex query (with 5 to 10 atomic conditions) that "make sense" for that source.

Figure 2.9 shows the results of our experiment. The running time of *GenCompact* is dominated by the parser calls (actually, calls to the *Check* function) it makes and the MCSC problems it solves. So, we measured the number of parser calls, the number of MCSC problems solved and the size of the input to the MCSC problems. In the figure, Q_{avg} refers to the average input size to the MCSC problems solved when processing the query, while Q_{max} refers to the maximum input size to the MCSC problems solved. Over all sources and queries studied, the average number of parser calls made per query was 24.5, the average size (number of nodes) of the condition tree parsed was 3.9, the average number of times the MCSC problems solved was 1.3 and the average input size (number of subplans) for the MCSC problems was 1.2. We also observed that the maximum input size to any MCSC problem over all sources and queries was 3. These numbers indicate that the work that *GenCompact* performs is very reasonable, at least for these "human-generated" queries. The effort is most likely insignificant when compared with that involved in executing the queries over the Web.

In a second scenario we considered a larger number of queries, but the queries were constructed randomly. We examined random queries of three different sizes: small (2 or

Source	Query	# of Parser Calls	$\|CT\|_{avg}$	# of MCSC Problems	Q_{avg}	Q_{max}
Amazon	q_a	1	4	0	0	0
(www.amazon.com)	q_b	14	3.14	2	2	3
Library of Congress	q_c	15	2.13	1	1	1
(lcweb.loc.gov)	q_d	16	3.12	2	1	1
Junglee	q_e	5	5.4	2	1.5	2
(www.junglee.com)	q_f	3	8.33	1	2	2
CQ	q_g	15	2.13	1	1	1
(conjunctive source)	q_h	127	3.50	1	1	1

Figure 2.9: Efficiency of *GenCompact* in Processing "Sensible" Queries.

3 atomic conditions), medium (about 5 atomic conditions), and large (about 10 atomic conditions). The larger queries may model situations where a front-end tool assists a human in generating queries.

We considered the same set of Web sources as before, and produced 10,000 random queries for each query size and source. Figure 2.10 shows the results of our experiment. Over all the sources, for small query-condition expressions, the average number of parser calls was 7.2, the average size of the condition tree parsed was 2.2, the average number of MCSC calls was 1.3 and the average size of the MCSC problem input was 1.4. For large expressions, the numbers were 85.8, 5.8, 4.2 and 1.6 respectively. Even though the number of parser calls increased noticeably, it is still quite reasonable given that the large queries have many atomic conditions (about 10).

In the random-queries scenario, we observed that the *maximum* MCSC size was much larger (93) than the average for one of the sources (Library of Congress). We believe that it is because this source is very powerful and so it yields many feasible subplans for parts of the target-query condition. In such cases, one may need to consider more efficient MCSC algorithms that solve the optimization problem approximately. The use of approximate solutions can cause *GenCompact* to generate suboptimal plans for the target queries. To understand this tradeoff, we conducted simple experiments involving a version of *GenCompact* that uses a very efficient greedy algorithm to solve the MCSC problems. We observed that not using an exhaustive algorithm has minimal effect on the cost of the query plans produced. The greedy *GenCompact* often produced plans that are within 3% of those produced by the exhaustive *GenCompact*.

Source	Type of Query	# of Parser Calls	$\|CT\|_{avg}$	# of MCSC Problems	Q_{avg}	Q_{max}
Amazon	small	7.6	2.1	1.4	1.2	3
(www.amazon.com)	medium	18.8	3.13	2.3	1.3	5
	large	87.3	5.7	4.6	1.4	7
Library of Congress	small	6.2	2.3	1.2	1.8	7
(lcweb.loc.gov)	medium	15.5	3.4	1.7	1.95	15
	large	80.8	6.1	3.4	2.2	93
Junglee	small	7.3	2.2	1.4	1.3	3
(www.junglee.com)	medium	18.3	3.2	2.3	1.4	5
	large	86.3	5.8	4.5	1.5	8
CQ	small	7.4	2.1	1.4	1.2	3
(conjunctive source)	medium	18.3	3.2	2.2	1.3	5
	large	88.7	5.8	4.4	1.4	8

Figure 2.10: Efficiency of *GenCompact* in Processing a Random Query.

2.5.2 Quality of *GenCompact* Plans

To gauge the quality of plans generated by *GenCompact*, we compared the plans it produced with the plans produced by the following two schemes:

1. *GenDNF*: The condition expression of the target query is transformed to DNF and a plan is generated independently for each term. The plan for each term evaluates as many of its atomic conditions as possible at the source, as in [57]. The plans for the terms are combined with a mediator union to arrive at the plan for the target query.

2. *GenCNF*: The condition expression of the target query is transformed to CNF and all the clauses that can be evaluated at the source are determined. All such clauses are evaluated at the source while the rest of the clauses are evaluated at the mediator.

GenDNF and *GenCNF* represent the kinds of techniques used to process target queries in contemporary systems like [28, 41, 55, 57]. Since the *GenCompact* version we studied in our experiments did not use the distributivity rewrite rule, it is possible for *GenDNF* and *GenCNF* to generate better query plans than *GenCompact*.

To compare the three schemes, we considered the same sources as in Section 2.5.1 and generated 30,000 random target queries (small, medium and large) for each of them. For each query and source, the plan generated by each scheme was determined, and its cost

Source	Query Type	*GenCompact*	*GenDNF*	*GenCNF*
Amazon	small	winner	40%	46%
(www.amazon.com)	medium	winner	60%	54%
	large	winner	90%	54%
Library of Congress	small	winner	30%	69%
(lcweb.loc.gov)	medium	3.2%	winner	86%
	large	0.6%	winner	91%
Junglee	small	winner	39%	41%
(www.junglee.com)	medium	winner	67%	52%
	large	winner	91%	47%
CQ	small	winner	13%	64%
(conjunctive source)	medium	winner	37%	71%
	large	winner	73%	59%

Figure 2.11: Comparing Quality of Plans.

obtained using the cost model of Section 2.4.2. The average cost over all the queries was calculated for each scheme and compared against that of the other schemes.

Figure 2.11 compares the average cost of the query plans produced by each scheme for each source and query combination. Each row marks the "winning" scheme that attained the lowest average cost. The columns of the losing schemes show how much the winning scheme improved on the cost of the losing schemes. For example, the plans for small *Amazon* queries generated by *GenCompact* take an estimated 40% of the time taken by the plans generated by *GenDNF*. *GenCompact* produced better plans in almost all the cases. On average, the cost of *GenCompact* plans was about 50% of that of *GenDNF* plans, and about 40% of that of *GenCNF* plans. These numbers only covered cases where all the three schemes produced a feasible plan. In our experiments, we observed that *GenCompact* found feasible plans in many more cases than *GenDNF* and *GenCNF*.

2.6 Discussion

So far, we have seen how *GenCompact* efficiently generates good feasible plans for selection queries. It uses a specific cost model suitable for the Web context, along with a powerful source-capability description language that can describe selection-query capabilities of Web sources very well. In this section, we will briefly discuss how *GenCompact* can adapt to other source-capability description languages and other cost models. We also consider the issue of doing efficient selection-query processing involving expensive attributes.

Other Capability Description Languages. Even though we chose a specific source-capability description language in this chapter, the dependency of *GenCompact* on this language is not very strong. *GenCompact* assumes that the capability-description language allows for an efficient implementation of the *Check* function in a manner that automatically accommodates commutative transformations of the condition expression passed to it. *GenCompact* can adapt easily to other languages that describe source capabilities [29, 57]. For example, in the case of RQDL [57], it is very easy to implement a *Check* function satisfying the assumptions of *GenCompact*.

Other Cost Models. The *GenCompact* scheme of Section 2.4 employed a specific cost model that is suitable for the Web context. This cost model led to a set of pruning rules that are used by *GenCompact* to achieve efficient plan-space exploration. We believe that *GenCompact* can easily adapt to many other cost models. First of all, some or all of the pruning rules used by *GenCompact* may well be valid in many cost models. In this case, *GenCompact* continues to be very efficient in finding good feasible plans for the target queries. If some of the pruning rules are not valid in a given cost model, *GenCompact* may choose to not employ those pruning rules. In this case, it still finds good feasible plans, even though its plan-generation process may be more expensive. Finally, one may choose to employ some of the pruning rules in *GenCompact* even though they are not valid in a cost model under consideration. It may be the case that the pruning rules still provide good heuristics that lead *GenCompact* to find good feasible plans in most situations, while maintaining an efficient plan-generation process.

Expensive Attributes. When expensive attributes are involved in selection queries, special care should be taken in executing source queries projecting these attributes. For instance, a target query may specify large attributes like images in its "projection list." It may not be prudent to import these expensive attributes of a data object to the mediator when the data object still needs to go through further filtering (with respect to the target-query condition). One simple approach that may often yield good plans in these situations is to adopt a two-phase query-processing strategy. In the first phase, only the key attributes of the answer objects are obtained. That is, we come up with a new target query based on the original target query by removing expensive attributes that are not key attributes. For the new target query *GenCompact* can be employed to arrive at an efficient feasible plan. In the second phase, we can send the key attributes to the data sources and obtain the expensive attributes of the answer objects. Note that this strategy works well only if the data objects at the sources have key attributes and if these attributes are not expensive themselves.

2.7 Related Work

Very few query-processing systems take into account source capabilities. Conventional systems such as System R [61], Ingres [67], DB2 [22], and NonStop SQL [63] assume homogeneous source capabilities. Multidatabase systems [45, 48, 64] address heterogeneity and autonomy, but still assume that "simple" selection queries are supported by all sources.

A few new systems like the Information Manifold [40, 41], Garlic [28, 55], and DISCO [35, 72] have directly addressed source capabilities. In the Information Manifold, one can only process conjunctive queries. That is, one cannot submit selection queries with arbitrary condition expressions involving $\wedge$'s and $\vee$'s. In contrast, we allow arbitrary condition expressions.

Garlic allows arbitrary condition expressions. The condition expressions are transformed into CNF and then each clause in the CNF expression is considered for evaluation at the source. If the source cannot evaluate a clause, it is evaluated by Garlic itself. As we illustrated in Section 2.1, Garlic can end up retrieving too much data from the sources. If none of the clauses in the CNF expression can be evaluated at the source, Garlic attempts to download the entire source. Such a plan is not only expensive, but it may not be supported by the source. In general, our approach examines more options than Garlic to discover efficient feasible plans.

In the DISCO system, no restrictions are placed on the condition expressions, and their framework allows sources to express disjunctive as well as conjunctive condition-processing capabilities. However, DISCO does not explore the possibility of splitting the target-query condition into parts when looking for feasible plans. Only those options in which the source is capable of answering the entire condition expression, or none at all, are considered. For instance, both the example queries of Section 2.1 cannot be executed by the DISCO system.

Other approaches like [30, 36] also deal with the efficient processing of selection queries. However, the focus of their work is on efficiently evaluating condition expressions involving expensive predicates on a given set of data objects. They do not deal with the issue of finding feasible plans for selection queries in the face of limited source capabilities.

Wrappers are often proposed for dealing with heterogeneous and limited-capability sources [29, 38, 73]. However, if wrappers are to provide generic relational capabilities for Web sources, then they need to implement a scheme like the one we described in Section 2.4. That is, when a wrapper receives a query, it must find the best way to execute the query at the underlying source. The challenge faced by the wrapper is is precisely the same problem that we have addressed in this chapter.

Source-capability description languages, different from the one we presented in Section 2.2.3, have been proposed in the literature [57, 72, 78]. Their focus is on describing the capabilities of sources with respect to processing conjunctive queries. For example [57] uses the relational query description language (RQDL). The CFG based language we employed in this chapter is not as powerful as RQDL in some aspects like describing the join capabilities of sources. But it is more powerful than RQDL when describing selection-query capabilities. For instance, CFGs can describe sources that evaluate disjunctive condition expressions while RQDL cannot. Since our focus in this chapter is on selection-query processing, we used the CFG based language to describe the query capabilities of sources.

2.8 Chapter Summary

In this chapter we studied the generation of efficient feasible plans for selection queries in the Web context. We employed a simple capability-description language based on context free grammars. We presented an efficient scheme called *GenCompact* that generates good feasible plans for processing selection queries over limited-capability data sources. We studied the time complexity and plan quality of *GenCompact* using sample queries on a few interesting Web sources.

Chapter 3

Optimizing Large Join Queries

3.1 Overview

In data-integration systems, mediators facilitate integrated access to data across multiple autonomous sources [79]. Examples of such systems are TSIMMIS [13], Information Manifold [37], Garlic [9], and DISCO [71]. In these systems, mediators translate user queries on integrated views into source queries and postprocessing operations on the source-query results. The translation process can be quite challenging when integrating a large number of autonomous sources.

One of the important challenges for integration systems is to deal with the diverse capabilities of sources in answering queries [28, 44, 57]. In this chapter, we focus our attention on the kind of queries that are the most expensive in mediation systems, *large join queries*. We develop and present efficient algorithms to find good plans for such queries.[1]

3.1.1 Capabilities-Based Plan Generation

We consider mediator systems, where users pose conjunctive queries over integrated views provided by the mediator. These queries are translated into conjunctive queries over the source views to arrive at *logical query plans*. The logical plans deal only with the content descriptions of the sources. That is, they tell the mediator which sources provide the relevant data and what postprocessing operations need to be performed on this data. The

[1]The material in this chapter is based on ideas that are developed by the author with Chen Li, Jeff Ullman and Hector Garcia-Molina. Some of these ideas had been presented in the conference paper [80].

logical plans are later translated into *physical plans* that specify details such as the order in which the sources are contacted and the exact queries to be sent. Our goal in this chapter is to develop algorithms that will translate a mediator's logical plan into an *efficient*, *feasible* (does not exceed the source capabilities) physical plan. We illustrate the process of translating a logical plan into a physical plan by an example.

EXAMPLE 3.1.1 Consider a scenario where we are looking to borrow books from a local library. Suppose, we want to find out the availability status of books that are highly rated and that pertain to a desired topic. In trying to gather the relevant information, consider three data sources that provide information about books. Let the first source, S_1, be a web site (like the Library of Congress) that provides a catalog of books with attributes like `title`, `author` and `ISBN`. Let the second source, S_2, be a different web site (like the one belonging to Amazon.com) where we can find information about books like pricing and customer ratings. For the purpose of this example, let this site provide just two attributes `ISBN` and `ratings`. The `ratings` attribute indicates the quality rating of the book that reflects the opinion of the customers of this site about the book. Let the third source, S_3, be yet another web site (like the one belonging to a public library) that provides information about the availability of books for borrowing. To keep the example simple, let this site have just two attributes `ISBN` and `status`.

Let the query restrictions offered by the three sites be as follows:

Source	Contents	Must Bind
S_1	`R(title, author, ISBN)`	either `title` or `author`
S_2	`S(ISBN, ratings)`	`ISBN`
S_3	`T(ISBN, status)`	`ISBN`

The *Must Bind* column indicates what attributes must be specified in order to successfully query a source. For instance, queries sent to S_1 must either provide the `title` or the `author` attribute.

Let a mediator offer an integrated view across the three source views:

Mediator View:
`Books(title,author,ISBN,ratings,status)`   `:- R(title, author,ISBN),S(ISBN,ratings),T(ISBN,status)`

Suppose, we ask a query to find all the "5-stars" rated books on "Databases" that are "Available" in the local library. Our query can be formulated as follows:

```
ans(author, ISBN) :- Books(''Databases'', author, ISBN, ''5-stars'',
                           ''Available'')
```

The mediator would translate this query to the logical plan:

```
ans(author, ISBN) :- R(''Databases'', author, ISBN), S(ISBN, ''5-stars''),
                     T(ISBN, ''Available'')
```

The logical plan states the information the mediator needs to obtain from the three sources and how it needs to postprocess this information. In this example, the mediator needs to join the results of the three source queries on the `ISBN` attribute. There are many physical plans that correspond to this logical plan (based on various join orders and join methods). Some of these plans are feasible while others are not. Here are three physical plans for this logical plan:

- Plan P_1: Send query R("Databases", *author*, $ISBN$) to S_1; send query $S(ISBN$, "5-stars") to S_2; and send query $T(ISBN$, "Available") to S_3. Join the results of the three source queries on the `ISBN` attribute and return the `ISBN` and `author` values to the user.

- Plan P_2: Send query R("Databases", *author*, $ISBN$) to S_1; for each returned ISBN index i, send a query to S_2 to get the `ratings` information and check if the rating is "5-stars"; for each ISBN index that has the "5-stars" rating, send a query to S_3 and find out the `status` information and check if the status is "Available"; if so, return i and the corresponding `author` information of the book to the user.

- Plan P_3: This plan is similar to P_2, except that we reverse the S_2 and S_3 queries. That is, after getting the list of ISBN indexes from S_1, we first go to S_3 and filter out all the books that are not "Available" and then filter the remaining list with respect to the ratings criterion tested at source S_2.

Among the three physical plans, the first one is not feasible because the queries to sources S_2 and S_3 do not provide a binding for the `ISBN` attribute. The other two plans are feasible. If the number of "Database" books available in the local library is smaller than the number of good ("5-stars" rating) books on "Databases" sold at the bookstore site, plan P_3 may be more efficient than plan P_2. Accordingly, the mediator may execute plan P_3. $\square$

As illustrated by the above example, mediators cannot simply translate user queries on integrated views to source queries and send out the source queries. A naive query-processing scheme of that nature would make the mediator miss many opportunities to find feasible query plans. In order to identify the feasible query plans, mediators must be able to *pass bindings* from some subgoals to the others when constructing physical plans to process their logical plans. The availability of the bindings required in order to process any particular subgoal depends on the sequence of subgoals that have been processed before the subgoal under consideration. We also noted, in the example, that different feasible plans may exist with different query-processing costs. In such a case, the mediator must endeavor to find efficient plans among the set of feasible plans.

In this chapter, we study the problem of finding efficient feasible physical plans for mediator logical plans. In particular, we focus on the processing of *join queries* over a set of source relations that have attribute-binding restrictions. We study the complexity of the query-optimization problem and develop algorithms for constructing good query plans. We show that the problem of finding the most efficient feasible plan is $\mathcal{NP}$-hard. We then develop two algorithms that can find good feasible plans rapidly using heuristics.

Our first algorithm uses a *greedy* approach where the overall query plan is constructed one subgoal at a time by choosing the next subgoal in the sequence in a greedy manner. Our second algorithm uses a *clustering* approach where the subgoals are grouped into various subsets. Subplans are created from each subset of the subgoals and then the overall query plan is stitched together from the various subplans.

Both of our algorithms use the technique of passing bindings that are accumulated as they process sets and sequences of subgoals ahead of the next set of subgoals or the next individual subgoal. Using this technique of passing bindings, both of our algorithms guarantee the finding of a feasible plan, if one exists. Also, both the algorithms quickly (in polynomial time) determine whether or not the given logical plan has a feasible physical plan. We study the analytical performance of our algorithms and present a linear bound on the margin by which our solution can miss the optimal plan. Previously known research had not achieved this combination of efficient algorithms that generate provably good plans. We also present results of the simulation experiments we have conducted to study the performance of our algorithms in a wide range of scenarios. Our experiments show that our algorithms are quite efficient and they often find optimal and near-optimal plans.

3.2 Framework

Let S_1, ..., S_m be m sources in an integration system. To simplify the presentation, we assume that sources provide their data in the form of relations. If sources have other data models, one could use *wrappers* [29, 73] to create the simple relational view of data. Each source is assumed to provide a single relation. If a source provides multiple relations, we can model it in our framework as a set of logical sources, all having the same physical source. Example 3.1.1 showed three sources S_1, S_2 and S_3 providing three relations R, S and T respectively.

A *query* to a source specifies atomic values to a subset of the source-relation attributes and obtains the corresponding set of tuples. A source supports a set of access templates on its relation that specify binding requirements for source queries.[2] A query on a source relation must satisfy one of the access templates supported by the source. Satisfying a template means that the query should input a superset of the attributes that are required to be bound. In Example 3.1.1, source S_2 had one access template: $S^{bf}(\texttt{ISBN}, \texttt{ratings})$, while source S_1 had two: $R^{bff}(\texttt{title}, \texttt{author}, \texttt{ISBN})$ and $R^{fbf}(\texttt{title}, \texttt{author}, \texttt{ISBN})$. Each access template corresponds to a set of query instances on the source relation.

Mediators provide integrated views on top of the source relations. Users pose queries to the mediator by specifying different values for some of the attributes of a mediator view. That is, users query the mediator in a manner that is similar to querying a source, except in the case of a mediator the queries are on the mediator views instead of the source relations.

User queries on mediator views are translated into logical plans that expand out the mediator views in terms of the source relations. The mediator logical plans are sets of conjunctive queries involving the source relations, with some of the attributes bound by the criteria set by the corresponding user queries. Typically, a single user query may end up with a logical plan that is a set of conjunctive queries (as mediator views may be defined as *unions* and *joins* of source relations). The answer to a user query is the union of the results of this set of conjunctive queries. In the case of Example 3.1.1, the user query specified

[2] We consider source-capabilities described as *bf* adornment patterns that distinguish bound (b) and free (f) argument positions [75]. The techniques developed in this chapter can also be employed to solve the problem of mediator query planning when other source capability description languages are used.

some of the attributes of the mediator view Books and it is translated into a logical plan that is a single conjunctive query. Note that the mediator view did not involve any unions and consequently we ended up with a single conjunctive query for the logical plan.

In order to find the best feasible plan for a user query, we assume that the mediator processes the logical plan one conjunctive query at a time (as in [41, 55]). That is, for each conjunctive query of a logical plan, the optimizer arrives at the best way to execute the conjunctive query. The plan for the user query is then pieced together from the plans for the various conjunctive queries of the logical plan. Thus, we reduce the problem of finding the best feasible plan for the user query to the problem of finding the best feasible plan for a conjunctive query in the logical plan. Therefore, from now on, we assume without loss of generality that a logical plan has a single conjunctive query over the source relations.

3.2.1 Cost Model

As in Chapter 2, we adopt a cost model that emphasizes the network costs. In particular, we choose the simple linear model we discussed in Section 2.4.2, where each source query has a fixed component of the cost and a variable component of the cost. The cost of a query q to source S_i is given by the expression:

$$Cost(q, S_i) = f_i + v_i * (size\ of\ result\ of\ q\ on\ S_i)$$

The cost of a query plan is the sum of the costs of the source queries (as before, we consider the local-processing costs at the mediator as negligible).

We illustrate our cost model by the following example.

EXAMPLE 3.2.1 Consider the data sources and the query plans of Example 3.1.1. Let each source query have a cost of 1 unit. Consider the query plan P_2. In this plan, we first query S_1 to find out all books on "Databases". Then, for each of these books we visit S_2 and S_3 to verify that the desired books are rated "5-stars" and are "Available" in the local library. Since we need to send only one query to S_1, the cost is 1 unit. Let there be 100 books that were written on the topic of "Databases." For each `ISBN` i, we send S_2 a query `S(i, ratings)` to get its `ratings`. Thus, the cost of processing the 100 queries is 100 units. If 10 of these 100 books had "5-stars" rating, we need to send 10 queries to S_3 to check the status of the book in the local library. So, the cost of the queries to S_3 is 10 units. Therefore, the total cost of the query plan $Cost(P_1) = 1 + 100 + 10 = 111$. □

3.2.2 Subgoals and Sequences

Let a mediator logical plan be $H :- C_1, C_2, \ldots, C_n$. We call each C_i a *subgoal*. Each subgoal specifies a *logical* query on one of the source relations by binding a subset of the attributes of the source relation. The bindings for the attributes come from the specification of the user query. We refer to the attributes of subgoals in the logical plan as *variables*. In Example 3.1.1, the logical plan had three subgoals with five variables, three of which were bound.

The essence of the mediator query processing is to execute the conjunctive queries involving the subgoals. However, as we noted earlier, it may not be possible to process the subgoals by simply sending the corresponding queries to the sources. Some of these source queries may be disallowed by the sources, if they do not meet their binding requirements. The mediator needs to consider the possibility of passing bindings across subgoals in order to construct feasible query plans.

When subgoals are processed by sending the source queries as they appear in the logical plan, we call the corresponding source queries *block queries*. When a subgoal is processed by passing additional bindings, computed from the processing of other subgoals, we call the corresponding source queries *parameterized queries*. We note that typically a subgoal that is processed in the latter way ends up having a set of parameterized queries. The distinguishing characteristic of the first kind of queries (i.e., block queries) is that a single query suffices to process the corresponding subgoal.

Note that a single parameterized query is usually cheaper than the corresponding block query because the parameterized query tends to bind more attributes of the source relation. However, processing a subgoal through a block query may be much cheaper than processing it through the corresponding set of parameterized queries because the number of parameterized queries tends to be large and thus the per-query fixed-cost factor dominates. The particular decision about each subgoal, whether it should be processed through a block query or parameterized queries, depends on the cost model. The decision is also influenced by the source-capability restrictions (the block-query option may be disallowed because the query does not bind the required attributes). When considering the processing of a subgoal through parameterized queries, one needs to take into consideration what attributes to parameterize. Correspondingly, the number of parameterized queries and the cost of each of these queries can be determined.

As discussed above, the feasibility of processing a subgoal and the cost of processing a subgoal are crucially dependent on the various source-query options (block query and different parameterizations) that are available. These options are directly dependent on the sequence of subgoals that have been processed ahead of the subgoal currently processed. Accordingly, various parameterizations of the source queries for the subgoal are possible. Thus, the key aspect of the query-planning process is to determine the sequence of subgoals. Once a sequence of subgoals is determined, the decision of which query/queries should be sent to the source for each subgoal can be determined efficiently in a local manner.

3.2.3 The Space of Query Plans

The space of all possible plans for a given user query is defined first by considering all sequences of subgoals in its logical plan. In a sequence, we must then decide on the choice of queries for each subgoal – the block query or various sets of parameterized queries. We call a plan in this space *feasible* if all the queries in it are allowed by the corresponding sources. The number of feasible physical plans, as well as the number of all plans, for a given logical plan can be exponential in the number of subgoals in the logical plan.

The optimization problem we are studying in this chapter is to find the most efficient feasible (i.e., *optimal*) physical plan in the space of query plans corresponding to a given logical plan.

Note that the space of plans we consider is similar to the space of left-deep-tree executions of a join query. As stated in the following theorem, we do not miss optimal plans by not considering bushy-tree executions.

Theorem 3.2.1 *We do not miss optimal plans by considering only left-deep-tree executions of the subgoals.* □

Proof: Without loss of generality we assume that a bushy-tree plan allows the passing of bindings from its left branch to its right branch at each of the join nodes. If there are nodes having right-to-left passing of bindings we can consider an equivalent bushy-tree plan that will switch the left and right branches of these nodes. Accordingly, we assume that the bushy plan is executed in a left-to-right manner.

Note that in our cost model the only cost that counts is that of the source queries. That is, the local-join operations at the mediator do not count. The cost of source queries depends on whether the corresponding subgoal is processed via a block query or a set of parameterized queries. If the subgoal is processed by parameterized queries then its cost depends on the set of variables that are parameterized and the number of value-sets of these parameters. This determination in turn is directly dependent on the sequence of subgoals preceding a particular subgoal whose cost is being evaluated. Given a set of variable bindings, the parameterization of source queries to process a subgoal can choose various subsets of the relevant variables.

Now, we show that there is no need to consider bushy-tree plans to find optimal plans. Given any bushy-tree plan, we consider a corresponding left-deep-tree plan that has the same leaf order as the bushy-tree plan. In other words, both these plans process the subgoals in the same sequence.

If a subgoal is processed in the bushy-tree plan via a block query, the corresponding subgoal can be processed in the left-deep-tree plan also via a block query. The cost remains the same in both cases. If a subgoal is processed in the bushy-tree plan via parameterized queries, the corresponding subgoal can be processed in the left-deep-tree plan also via parameterized queries. In fact, the same set of variables can be parameterized. There are two important issues that arise here. First, we are guaranteed to have the required variable bindings available in the left-deep-tree plan because it has a sequence of subgoals (preceding the subgoal being processed) that includes all the subgoals of the bushy-tree plan from which the parameters are passed in the bushy-tree plan. In other words, *the bindings are preserved.* Second, the only difference between the left-deep-tree plan and the bushy-tree plan as far as the particular subgoal being processed is concerned is that in the left-deep-tree plan all the preceding subgoals are joined to form the bindings for the variables being parameterized while in the bushy-tree plan a subset of these subgoals may be joined together to form the bindings. The former is guaranteed to be a subset of the latter. That is, by joining a superset of the subgoals, the left-deep-tree subgoal execution is guaranteed to generate a subset of the queries that the bushy-tree plan generates for processing this subgoal. Thus, the cost of the subgoal in the left-deep-tree plan cannot be higher than that of the bushy-tree plan.

We have shown above that the cost of each subgoal in the left-deep-tree plan cannot be larger than the corresponding cost in the bushy-tree plan (whether the subgoal is processed via a block query or a set of parameterized queries). Therefore, the cost of the left-deep-tree plan cannot be higher than the bushy-tree plan. Since we can construct a left-deep-tree plan that is at least as cheap as any given bushy-tree plan, we will not miss the optimal plan by confining our search to left-deep-tree plans. ∎

It is relatively easy to construct the actual query plans after determining the *correct* sequences of subgoals. Hence, we focus much of our attention in the rest of this chapter on finding optimal sequences of subgoals. The algorithms we develop in later sections construct optimal sequences of subgoals and then convert these sequences into the corresponding plans.

Sequences of subgoals satisfy some interesting and useful properties stated by the following lemmas.

Lemma 3.2.1 *Postponing the processing of an answerable subgoal in a sequence can not make it unanswerable.* □

Proof: A subgoal is either answerable through a block query or a set of parameterized queries. In the former case, the subgoal is always answerable no matter where it is processed in the sequence. In the latter case, postponing the subgoal cannot make it unanswerable because the set of variables that need to be bound in the parameterized queries to the source remain bound at a later point in the sequence. The actual values of the bindings for the variables may change (i.e., the number of different sets of values, each set leading to a parameterized query, can shrink), but the variables remain bound. Thus, if a subgoal is answerable at a certain point in the sequence, it remains answerable if it is postponed in the sequence. ■

Lemma 3.2.2 *Postponing the processing of an answerable subgoal in a sequence can not increase its cost.* □

Proof: The proof is along similar lines of Lemma 3.2.1. That is, consider a subgoal that is answerable at a certain point in a sequence. If it is answered using a block query, it can also be answered using the same block query at a later point in the sequence. The cost of processing the subgoal remains the same. If it is answered using a set of parameterized queries, it can also be answered at a later point in the sequence using another set of parameterized queries based on the same set of bound variables. The actual set of parameterized queries sent to the source to process the subgoal may differ. This set depends on the different sets of values for the set of bound variables. Since the sets of values for a set of bound variables can only shrink at a later point in the sequence, the set of parameterized queries sent to process the subgoal at a later point in the sequence is a subset of the set of parameterized queries used to answer the subgoal at its original point in the sequence. Thus, the cost of answering the subgoal cannot increase if the subgoal is postponed in a sequence. ■

3.2.4 Complexity of the Problem

Theorem 3.2.2 *The problem of finding optimal physical plans for mediator logical plans is $\mathcal{NP}$-hard.* □

Proof: We reduce the Vertex Cover problem [21] to our problem. Since the Vertex Cover problem is known to be $\mathcal{NP}$-complete, our problem is $\mathcal{NP}$-hard.

Given a graph G with n vertices $V_1, \ldots, V_n$, we construct a database and a logical plan as follows. Corresponding to each vertex V_i we define a relation R_i. For all $1 \leq i \leq j \leq n$, if V_i and V_j are connected by an edge in G, R_i and R_j include the attribute A_{ij}. In addition, we define a special attribute X and two special relations R_0 and R_{n+1}. In all, we have a total of $m+1$ attributes, where m is the number of edges in G. The special attribute X is in the schema of all the relations. The special relation R_{n+1} also has all the attributes A_{ij}. That is, R_0 has only one attribute and R_{n+1} has $m+1$ attributes. Each relation has a tuple with a value of 1 for each of its attributes. In addition, all relations except R_{n+1} include a second tuple with a value of 2 for all their attributes. Each relation has a single access template: R_0 has no binding requirements, R_1 through R_n require the attribute X to be bound, and R_{n+1} requires all of the attributes to be bound. Finally, the logical plan consists of all the $n+2$ relations, with no variables bound.

It is obvious that the above construction of the database and the logical plan takes time that is polynomial in the size of G. Now, we show that G has a vertex cover of size k if and only if the logical plan has a feasible physical plan that requires $(n+k+3)$ source queries.

Suppose G has a vertex cover of size k. Without loss of generality, let it be $V_1, \ldots, V_k$. Consider the physical plan P that first answers the subgoal R_0 with a block query, then answers $R_1, \ldots, R_k, R_{n+1}, R_{k+1}, \ldots, R_n$ using parameterized queries. P is a feasible plan because R_0 has no binding requirements, $R_1, \ldots, R_k$ need X to be bound and X is available from R_0, and $R_1, \ldots, R_k$ will bind all the variables (since $V_1, \ldots, V_k$ is a vertex cover). In P, R_0 is answered by a single source query, $R_1, \ldots, R_k$ and R_{n+1} are answered by two source queries each, and $R_{k+1}, \ldots, R_n$ are answered by one source query each. We have a total of $(n+k+3)$ source queries for this plan. Thus, we see that if G has a vertex cover of size k, we have a feasible plan with $(n+k+3)$ source queries.

Suppose, there is a feasible plan P' with f source queries. In P', the first relation must be R_0, and this subgoal must be answered by a block query (because the logical plan does not bind any variables). All the other subgoals must be answered by parameterized queries. Consider the set of subgoals in P' that are answered before R_{n+1} is answered. Let j be the size of this set of subgoals (excluding R_0). Since R_{n+1} needs all attributes to be bound, the union of the schemas of these j subgoals must be the entire attribute set. That is, the vertices corresponding to these j subgoals form a vertex cover in G. In P', each of these j subgoals takes two source queries, along with R_{n+1}, while the rest of $(n-j)$ subgoals in $R_1, \ldots, R_n$ take one source query each. That is, $f = 1 + 2 * j + 2 + (n-j)$. Therefore, we can find a vertex cover for G of size $(f-n-3)$. Hence, G has a vertex cover of size k if and only if there is a feasible plan with $(n+k+3)$ source queries.

Thus, we have reduced the problem of finding the minimum vertex cover in a graph to our problem of finding a feasible plan with minimum source queries. ∎

3.3 A Greedy Approach

In this section, we present an algorithm, called CHAIN, for finding the best feasible physical plan for a given logical plan. Algorithm CHAIN is based on a simple greedy strategy of building an efficient feasible physical plan one subgoal at a time. In each step, CHAIN looks to add the next subgoal from among those that are yet to be processed and that are feasible with respect to the set of variable bound by the subgoals that are already processed.

The formal description of the algorithm is presented in Figure 3.1. As shown in the figure, algorithm CHAIN starts by identifying all the subgoals that are answerable with the initial bindings specified in the logical plan. It chooses the cheapest subgoal among these and constructs a sequence starting with it. When a subgoal is chosen, the set of bound variables is updated (i.e., the variables newly bound due to the processing of this subgoal are added to the set). Then, algorithm CHAIN proceeds to find the next subgoal in the sequence. It repeats this process of adding the next cheapest feasible subgoal, until no more subgoals can be added. When CHAIN terminates, either it is the case that there are no more subgoals left to consider or it is the case that none of the subgoals that are considered can be answered using the variables bound by the subgoals already included in the sequence constructed by CHAIN. In the first case, CHAIN successfully constructs a feasible physical plan and returns the plan. In the second case, CHAIN declares that no feasible physical plan exists for the given logical plan.

When considering the task of picking the cheapest feasible subgoal to be added to the sequence, algorithm CHAIN uses a function named COST. This function computes the cost of a feasible subgoal with respect to a set of bound variables. In essence, it determines the best way to process the subgoal given the variable bindings computed by the partially constructed sequence of subgoals collected so far.

3.3.1 Quality of Plans Generated by CHAIN

CHAIN is a very efficient algorithm. Its worst-case time complexity is a low-degree polynomial in terms of its input size. Moreover, CHAIN is guaranteed to find feasible plans when they exist. Finally, CHAIN has the attractive property that the plans generated by it cannot be arbitrarily far from the optimal plans. The following lemmas and theorems present formally the facts about CHAIN's quality of plans.

Lemma 3.3.1 *CHAIN will not miss feasible physical plans.* □

Algorithm 3.3.1 *CHAIN*
Input: Set of subgoals $G = \{C_1, C_2, \ldots, C_n\}$; set of bound variables B
Output: Feasible physical plan, if one exists
Method:

```
S ← [ ]
While (G ≠ {})
    M ← infinity
    For each subgoal C_i in G
        If (C_i is answerable with B)
            c ← COST(S, C_i)
            if (c < M)
                M ← c
                N ← C_i
    If (M == infinity)
        Return({})
    S ← S · [ N ]
    G ← G − {N}
    B ← B ∪ {variables of N}
Return(Plan(S)) ◇
```

Figure 3.1: Algorithm *CHAIN*.

$R^{bff}(A,B,D)$	$S^{bf}(B,E)$	$T^{bf}(D,F)$
(1, 1, 1)	(1, 1)	(4, 1)
(1, 2, 2)	(2, 1)	(5, 1)
(1, 3, 3)	(3, 1)	(6, 1)
(1, 1, 4)	(4, 1)	(7, 1)

Table 3.1: Database Instance: R, S and T.

Proof: The only situation in which CHAIN fails to generate a feasible physical plan is when it is left with a set of subgoals none of which can be answered using the initial bindings and the bindings of the variables appearing in the subgoals already processed. In such a case, there cannot be a feasible physical plan for the logical plan. If there were a feasible physical plan in this case, consider the first subgoal in the feasible physical plan that is one of the leftover subgoals in CHAIN. That is, all the subgoals preceding this subgoal in the feasible plan are in the sequence built by CHAIN so far. This subgoal will be identified as answerable by CHAIN because it will have a set of bound variables that would be sufficient to cover the bindings used to process this subgoal in the feasible physical plan. Hence, CHAIN would have added this subgoal to its sequence. Therefore, CHAIN cannot end up declaring that there is no feasible plan, when such a plan actually exists. ∎

Algorithm CHAIN does not guarantee optimal plans in all cases. In fact, even in simple cases involving a small number of subgoals, it does not guarantee optimal plans, as illustrated by the following example.

EXAMPLE 3.3.1 Consider a logical plan $H: -R(1,B,D),\ S(B,E),\ T(D,F)$ with three subgoals R, S and T. Let the access templates be *bff*, *bf* and *bf* respectively. Let the cost of each source query be 1 unit. Let the data in R, S and T be as shown in Table 3.1.

Since the only variable bound by the user query is A, CHAIN will start the sequence with R. Then, both S and T become answerable (because R bounds B and D). The cost of S is 3 (B has 3 values) and the cost of T is 4 (D has 4 values). So, it choose S. Thus, CHAIN ends up with the sequence $R \rightarrow S \rightarrow T$. The cost of the plan based on this sequence is $1+3+4=8$.

Now, consider another feasible plan based on the sequence $R \rightarrow T \rightarrow S$. The cost of this plan is $1+4+1=6$. The crucial point here is that after R and T are processed the number of B values goes down from 3 to 1. So, processing T (the more expensive subgoal) ahead of S yields a benefit of 2 that CHAIN does not take into consideration.

Thus, CHAIN misses the optimal plan. □

It is not difficult to find situations in which CHAIN misses the optimal plan. However, surprisingly, there is a linear upper bound on how far its plan can be from the optimal. In other words, no matter what sources and logical plan we are given, algorithm CHAIN never performs too badly.

Lemma 3.3.2 *For a logical plan with n subgoals, let P^c be the plan generated by CHAIN, P^o be an optimal plan, and L be the cost of the most expensive subgoal in P^o. Then,*

$$Cost(P^c) \leq n \times L$$

. $\square$

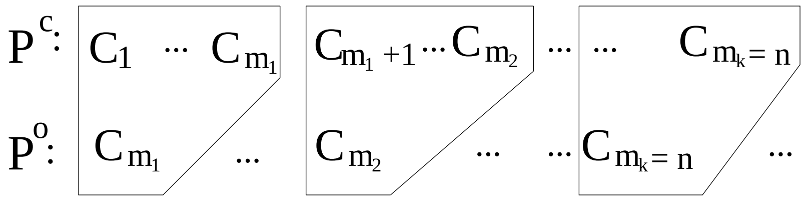

Figure 3.2: Proof of Lemma 3.3.2.

Proof: Without loss of generality, suppose the sequence of subgoals in P^c is $C_1, C_2, \ldots, C_n$. As shown in Figure 3.2, let the first subgoal in P^o be C_{m_1}. Let S_1 be the prefix of P^c, such that $S_1 = C_1 \ldots C_{m_1}$. When CHAIN chooses C_1, the subgoal C_{m_1} is also answerable. Therefore, the cost of C_1 in P^c is less than or equal to the cost of C_{m_1} in P^o. After processing C_1 in P^c, the subgoal C_{m_1} remains answerable (see Lemma 3.2.1) and its cost of processing cannot increase (see Lemma 3.2.2). So, if CHAIN has chosen another subgoal C_2 instead of C_{m_1}, once again we can conclude that the cost of C_2 in P^c is not greater than the cost of C_{m_1} in P^o. Finally, at the end of S_1, when C_{m_1} is processed in P^c, we note that the cost of C_{m_1} in P^c is no more than the cost of C_{m_1} in P^o. Thus, the cost of each subgoal of S_1 is less than or equal to the cost of C_{m_1} in P^o.

We call C_{m_1} the first *pivot* in P^o. We define the next pivot C_{m_2} in P^o as follows. C_{m_2} is the first subgoal after C_{m_1} in P^o such that C_{m_2} is not in S_1. Now, we can define the next subsequence S_2 of P^c such that the last subgoal of S_2 is C_{m_2}. The cost of each subgoal in S_2 is less than or equal to the cost of C_{m_2}.

We continue finding the rest of the pivots $C_{m_3}, \ldots C_{m_k}$ in P^o and the corresponding subsequences $S_3, \ldots, S_k$ in P^c. Based on the above argument, we have:

$$\forall C_i \in S_j : (\text{cost of } C_i \text{ in } P^c) \leq (\text{cost of } C_{m_j} \text{ in } P^o)$$

Consequently,

$$Cost(P^c) = \sum_{j=1}^{k} \sum_{C_i \in S_j} (\text{cost of } C_i \text{ in } P^c) \leq \sum_{j=1}^{k} |S_j| \times (\text{cost of } C_{m_j} \text{ in } P^o) \leq n \times L$$

∎

Theorem 3.3.1 *CHAIN is n-competitive. That is, the plan generated by CHAIN can be at most n times as expensive as the optimal plan, where n is the number of subgoals.* □

Proof: Since the cost of the most-expensive subgoal in the optimal plan is clearly not greater than the entire cost of optimal plan, it follows from Lemma 3.3.2 that the plan generated by algorithm CHAIN can at most be n times as bad as the optimal plan. ∎

Theorem 3.3.1 sets an upper bound on the cost of the plans generated by algorithm CHAIN. One may wonder how tight this upper bound is. It turns out that the bound is very tight. That is, the cost of the plan generated by CHAIN can be arbitrarily close to the cost of the optimal plan multiplied by the number of subgoals. The reason for the tight bound is that L (the cost of the most expensive subgoal) can be arbitrarily close to the entire cost of the optimal plan.

Even though the linear bound on optimality for CHAIN is tight and CHAIN can achieve results that are n times as bad as optimal, in many practical situations CHAIN yields optimal plans or plans whose cost is very close to that of the optimal plans. Section 3.5 discusses the performance of algorithm CHAIN in a wide range of practical scenarios.

3.4 A Clustering Approach

In this section, we present another algorithm called PARTITION that finds efficient feasible plans for processing mediator logical plans. PARTITION takes a very different approach, compared with CHAIN, to solve the plan-generation problem.

As shown in Figure 3.3, algorithm PARTITION has two phases. In the first phase, it starts with the variables that are bound initially in the query. It then collects all the subgoals that can be directly processed into a *cluster*. Based on the subgoals in this cluster, PARTITION computes the larger set of bound variables. Using this larger set of bound variables, it collects a new set of subgoals (from the set of subgoals not already collected previously) into a new cluster. PARTITION continues in this fashion until no more new

Algorithm 3.4.1 *PARTITION*
Input: Set of subgoals $G = \{C_1, C_2, \ldots, C_n\}$; set of bound variables B
Output: Feasible physical plan, if one exists
Method:

Phase I:

$\Psi \leftarrow \{\}$
While ($G \neq \{\}$)
  $N \leftarrow \{\}$
  For each subgoal C_i in G
    If (C_i is answerable with B)
      $N \leftarrow N \ \cup \ \{C_i\}$
  If ($N == \{\}$)
    Return($\{\}$) /* no feasible plan exists */
  $\Psi \leftarrow \Psi \ \cup \ \{N\}$
  $G \leftarrow G \ - \ N$
  $B \leftarrow B \ \cup \ \{$variables in subgoals of $N\}$

Phase II:

$S \leftarrow [\ ]$
For each cluster N in Ψ
  $S' \leftarrow$ the best subsequence of subgoals in N
  $S \leftarrow S \ \cdot \ S'$
Return($Plan(S)$) $\diamond$

Figure 3.3: Algorithm *PARTITION*.

clusters can be formed. There are two cases of termination. In the first case, all the subgoals of the query have been collected into some cluster and no more subgoals are left to consider. In the second case, some of the subgoals are not yet collected, but they cannot form a new cluster because they need variable bindings that are not available. In the first case, PARTITION will move onto the second phase and generate an efficient feasible plan for the query. In the second case, PARTITION terminates and declares that no feasible plan for the query can be found.

Clusters of subgoals are formed in the first phase of PARTITION in such a way that all the subgoals in the first cluster can be processed by block queries, all the subgoals in the second cluster can be processed by parameterized queries using bindings that result from processing the subgoals of the first cluster, all the subgoals in the third cluster can be processed by parameterized queries using bindings that result from the processing of the subgoals in the first two clusters, and so on. Accordingly, in the second phase, PARTITION finds the best subplan for each cluster of subgoals, by considering the variables bound by the previous clusters. In order to find the best subplan for each cluster, PARTITION performs local optimization by trying all the possible permutations of the subgoals in the cluster and the various binding patterns for the parameterized queries. The overall plan for the user query is obtained by combining the subplans for the various clusters.

EXAMPLE 3.4.1 Consider a logical plan with four subgoals:

$$H\ :-\ R(X,A,B),\ S(A,D),\ T(B,E),\ U(D,B)$$

Let the user query only specify a binding for variable X. Let the access templates of the four subgoals be *bff*, *bf*, *bf* and *bf* respectively. That is, to answer the subgoal on R the source query needs to bind X, to answer the subgoal on S the source query needs to bind A, etc.

Let us consider the first phase of the PARTITION algorithm. In the first cluster, PARTITION collects R. Processing R will yield two new bound variables A and B. Consequently, two new subgoals S and T become answerable. Thus, PARTITION forms the second cluster as $\{S,T\}$. The processing of the subgoals in the second cluster yields two more bound variables D and E. Accordingly, U is collected into the third cluster, as it is now answerable using the bindings of D. At the end of the first phase, PARTITION ends up with three clusters: $[\ \{R\},\{S,\ T\},\{U\}\ ]$.

In the second phase, the processing of the first and third cluster is trivial because they have only one subgoal each. For the second cluster, PARTITION considers two feasible subsequences of subgoals, and chooses the one with the lower cost, say $S \rightarrow T$. The overall sequence of subgoals constructed by PARTITION would then be $R \rightarrow S \rightarrow T \rightarrow U$. □

3.4.1 Feasible Plan Generation

Lemma 3.4.1 *Algorithm PARTITION does not miss feasible plans.* □

Proof: Given a logical plan (i.e., a set of subgoals and a set of bound variables), PARTITION determines that no feasible plan exists only when some of the subgoals are not answerable with respect to the set of variables that are initially bound or that occur in the other subgoals. If there

$R^{bff}(X, A, B)$	$S^{bf}(A, D)$	$T^{bf}(B, E)$	$U^{bf}(D, B)$
(1, 1, 1)	(1, 1)	(1, 1)	(1, 1)
(1, 1, 2)			
(1, 1, 3)			
...			
(1, 1, 10000)			

Table 3.2: Database Instance: R, S, T, U.

is a feasible physical plan, it is not possible to have such unanswerable subgoals in the logical plan. Hence, PARTITION will not fail to find a feasible plan when feasible plans exist. ∎

Unlike CHAIN, PARTITION does not have any guarantee of bounded optimality. The following lemma states this fact and proves it formally.

Lemma 3.4.2 *For any $k > 0$, there exists a logical plan and a database for which PARTITION generates a plan that is at least k times as expensive as the optimal plan.* □

Proof: Consider the logical plan of Example 3.4.1 (with four subgoals). Suppose the database instance at the four sources is as shown in Table 3.2. As before, PARTITION ends up with the following clusters: $\{R\}$, $\{S, T\}$, $\{U\}$. Given that there are two subgoals in the second cluster, the plan space of PARTITION contains two subgoal sequences: $R \rightarrow S \rightarrow T \rightarrow U$, $R \rightarrow T \rightarrow S \rightarrow U$.

Suppose the cost of each source query is 1 unit. The cost of the plan based on the first sequence is $1+1+10000+1 = 10003$. The cost of the plan based on the second sequence is $1+10000+1+1 = 10003$. PARTITION can choose between these two plans. Therefore, it will end up with a plan that costs 10003.

Now consider the feasible sequence $R \rightarrow S \rightarrow U \rightarrow T$. The cost of a physical plan based on this sequence is $1 + 1 + 1 + 1 = 4$. Essentially, by answering U before T, we can save on the cost of answering T (by avoiding many parameterized queries with B values other than 1). PARTITION misses this plan because it processes all subgoals in a cluster before it considers subgoals from the next cluster. Since we found a plan with 4 units of cost, the ratio of the cost of the PARTITION plan to the optimal cost is at least 10003/4. We can make this ratio arbitrarily large by having the appropriate number of tuples in R. Thus, PARTITION can generate plans that are at least k times as expensive as optimal plans, for any $k > 0$. ∎

3.4.2 Comparison with CHAIN

As we have seen, PARTITION can miss the optimal plans by arbitrary margins, while CHAIN has guaranteed bounded optimality. However, in some cases PARTITION can be better than CHAIN. That is, it can find optimal plans in cases where CHAIN misses. At the same time, it can miss optimal plans where CHAIN can find them. The following lemmas formally compare the abilities of PARTITION and CHAIN in finding optimal plans.

Lemma 3.4.3 *There are cases in which PARTITION generates optimal plans while CHAIN misses the optimal plans.* □

Proof: Consider the logical plan of Example 3.3.1. That plan had three subgoals R, S and T with access templates bff, bf and bf respectively. CHAIN misses the optimal sequence $R \rightarrow T \rightarrow S$, and erroneously picks the sequence $R \rightarrow S \rightarrow T$. For this query, PARTITION ends up with two clusters $\{R\}$ and $\{S, T\}$. In its second phase, it considers both of the above sequences of subgoals and hence finds the optimal one. ■

Lemma 3.4.4 *There are cases in which PARTITION misses the optimal plans while CHAIN generates the optimal plans.* □

Proof: Consider the example instance of the database and the query of Lemma 3.4.2. PARTITION ends up with three clusters: $\{R\}$, $\{S, T\}$ and $\{U\}$. In its second phase, it considers two sequences: $R \rightarrow S \rightarrow T \rightarrow U$ and $R \rightarrow T \rightarrow S \rightarrow U$. Neither of these two sequences is optimal. The optimal sequence is $R \rightarrow S \rightarrow U \rightarrow T$. CHAIN first chooses R as it is the only feasible subgoal with the query bindings. Then S and T are identified as candidates for the next subgoal. S has a cost of 1 unit and T has a cost of 10000 units. So, CHAIN picks S. With the processing of S, subgoal U becomes feasible and competes with T to be the next subgoal. The cost of U is 1 and the cost of T is 10000, so U wins. Thus, CHAIN ends up with the optimal sequence: $R \rightarrow S \rightarrow U \rightarrow T$. ■

3.5 Performance Analysis

We have seen that neither CHAIN nor PARTITION can guarantee optimal plans. CHAIN has guaranteed bounds on optimality but the bounds are not very attractive. PARTITION does not have any bound on how far away from optimal its plans can be. Given these facts, we need to understand how well these algorithms perform in practice? That is, how often do they find optimal plans? And how much margin of error do they have with respect to optimal plans? We have established, in Section3.4, that CHAIN can be better than PARTITION in some cases and PARTITION can be better than CHAIN in some other

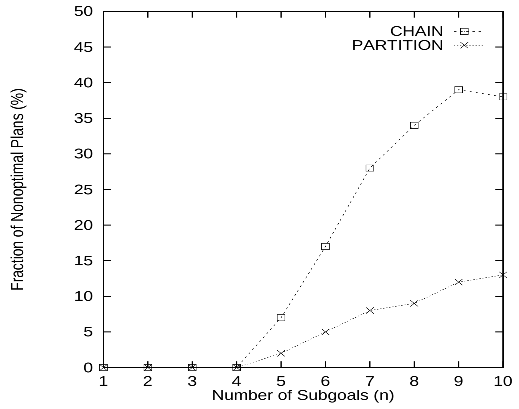

Figure 3.4: Fraction of Non-optimal Plans.

cases. We would like to understand the relative performance of PARTITION of CHAIN in practice. In this section, we present results of various experiments we have conducted that provide valuable insights to answer the above questions.

3.5.1 Simulation Environment

We started with a set of 15 data sources. We constructed two access templates for each source. Each of the access templates required one attribute to be bound. We populated the sources with a size distribution of 30% small, 60% medium and 10% large.

We implemented the CHAIN and PARTITION algorithms and fed them a series of queries and studied the performance of the plans constructed by the algorithms. We chose queries that involved random subsets of the 15 sources. The number of subgoals in each query varied from 1 to 10. We also exhaustively searched for the optimal plan for each of the queries. We computed the costs of the plans, constructed by CHAIN, PARTITION and the exhaustive optimal-plan generator, based on a simplified cost model where the cost of each source query is 1 unit.

3.5.2 Finding Optimal Plans

We conducted experiments that measured the fraction of the queries for which the algorithms found optimal plans. For each number of subgoals, varying from 1 to 10, we generated 100 random queries and studied the performance of CHAIN and PARTITION with respect to their ability to find the optimal plans.

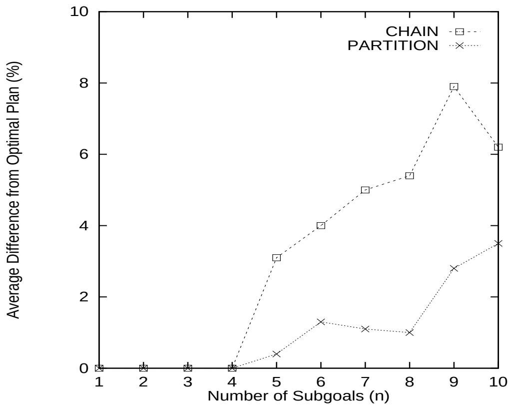

Figure 3.5: Average Cost.

Figure 3.4 shows how often CHAIN and PARTITION found optimal plans. Over the entire set of 1000 queries with number of subgoals ranging from 1 to 10, CHAIN found the optimal plans more than 80% of the time, while PARTITION found the optimal plans more than 95% of the time. This result is surprising because we had proved in Section 3.3 that logical plans with as few as 3 subgoals can lead CHAIN to miss the optimal plans, and logical plans with as few as 4 subgoals can lead PARTITION to suboptimal plans.

3.5.3 Margin of Error from Optimal Plans

Figure 3.5 plots the number of query subgoals (on the horizontal axis) vs. the average margin by which the plans generated by CHAIN and PARTITION miss the optimal plans (on the vertical axis). The average miss margin, over the entire range of queries (from 1 to 10 subgoals), is less than 10%. This fact demonstrates that CHAIN and PARTITION have excellent practical performance, even though PARTITION does not guarantee bounded optimality and CHAIN's bounds on optimality are not theoretically attractive.

Figure 3.6 shows the largest margin by which CHAIN and PARTITION miss the optimal plans. In the worst case, over the 1000 queries, CHAIN generated a plan that was less than twice as expensive as the optimal plan; the worst-case miss for PARTITION was also a plan that was slightly less than twice as expensive as the optimal plan. Once again, these results are surprising in that our theoretical results predicted that CHAIN can generate plans that cost as much as 10 times the optimal plan. We also proved that PARTITION can miss the optimal plan by an "unbounded" margin.

In summary, our experiments show that the PARTITION algorithm can have excellent

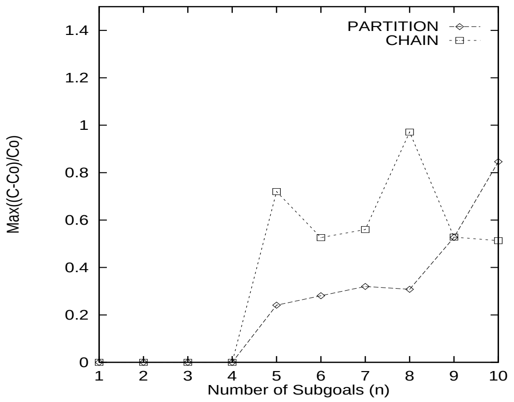

Figure 3.6: Worst-case Miss.

practical performance, even though it gives very few theoretical guarantees. The CHAIN algorithm also has very good performance, well beyond the theoretical guarantees we proved in Section 3.3. Finally, comparing the two algorithms, we observe that PARTITION consistently outperforms CHAIN in finding near-optimal plans.

3.6 Related Work

The central problem we have discussed in this chapter, namely the problem of ordering subgoals to find the best feasible sequence, can be viewed as the well known *join-order* problem [76]. More precisely, we can assign *infinite* cost to infeasible sequences of subgoals and proceed to find the best join order.

The join-order problem has been extensively studied in the literature, and many solutions have been proposed. Some solutions perform a rather exhaustive enumeration of plans, and hence do not scale well [3, 6, 15, 16, 28, 44, 51, 52, 57, 58, 62, 77]. In particular, we are interested in scenarios, like on the Web, with many sources and subgoals. So, these schemes are too expensive. Some other solutions reduce the search space through techniques like *simulated annealing*, *random probes*, or other heuristics [19, 33, 34, 50, 60, 69, 70]. While these approaches may generate efficient plans in some cases, they do not have any performance guarantees in terms of the quality of plans generated (i.e., the plans generated by them can be arbitrarily far from the optimal one). Many of these techniques may even fail to generate a feasible plan, while the user query does have a feasible plan.

The remaining solutions [32, 39, 66] use specific cost models and clever techniques that exploit them to produce optimal join orders efficiently. While these solutions are very good for the join-order problem where those cost models are appropriate, they are hard to adopt in our context because of two difficulties. The first difficulty is that it is not clear how to model the feasibility of mediator query plans in their frameworks. A direct application of their algorithms to the problem we are studying may end up generating infeasible plans, when a feasible plan exists. The second difficulty is that when we use cost models that emphasize the main costs in mediator systems, the optimality guarantees of their algorithms may not hold.

3.7 Chapter Summary

In this chapter, we considered the problem of query optimization in heterogeneous data-integration systems based on the mediation approach. We employed a cost model that focuses on the main costs in mediation systems. In this cost model, we developed two algorithms that guarantee the generation of feasible plans (when they exist). We showed that the problem at hand is $\mathcal{NP}$-hard. One of our algorithms runs in polynomial time. It generates optimal plans in many cases and in other cases it has a linear bound on the worst-case margin by which it misses the optimal plans. The second algorithm finds optimal plans in more scenarios, but has no bound on the margin of missing the optimal plans in the bad scenarios.

Chapter 4

Fusion-Query Optimization

4.1 Overview

In distributed information environments like the Web, one often encounters sources that provide incomplete and possibly overlapping information on a set of entities. A *fusion query* searches over these entities, looking for ones that satisfy given conditions. To illustrate, consider databases operated by the Departments of Motor Vehicles (DMVs) of several states. Conceptually, each state database can be thought of as a relation R_i with the following attributes, among others:

- Driver's license number (L): includes state and number within that state.
- Violation (V): A code representing an infraction, for example "driving under the influence" (*dui*) or "speeding" (*sp*).
- Date (D): the date of the infraction.

Figure 4.1 shows some sample relations for three DMVs. Now, consider a fusion query that searches for drivers who have both a *dui* and a *sp* violation. For instance, the driver with license *J55* satisfies this query because he has a *dui* infraction in the first state and a *sp* one in the second state. License *T21* also meets this criterion, but the others in Figure 4.1 do not. Notice that the information for a particular driver, say *J55*, may be dispersed among the sources, so the query conceptually (but not actually) "fuses" the information for each driver as it checks the constraints. Such a query is inherently expensive: we do not know in advance which sources may have the information fragments for *J55*, so the search must cover all possibilities.

R_1

L	V	D
J55	dui	1993
T21	sp	1994
T80	dui	1993

R_2

L	V	D
T21	dui	1996
J55	sp	1996
T11	sp	1993

R_3

L	V	D
T21	sp	1993
S07	sp	1996
S07	sp	1993

Figure 4.1: DMV Example.

To express our sample query in SQL, we first let U be the union of all the R_1, R_2, ... tables at the various DMVs, and then we write

```
SELECT  u1.L
FROM    U u1, U u2
WHERE   u1.L = u2.L
        AND  u1.V = sp AND u2.V = dui
```

In this chapter, we focus on developing techniques that can be used by mediators to process fusion queries.[1]

We believe that such fusion queries are important now, and will become even more important in the future as mediation systems cope with more and more information that has *not* been nicely structured and partitioned in advance. In a traditional distributed-database setting, for instance, an administrator could determine in advance that all violations for licenses issued in a given state go to a particular database. Such a design makes query processing much simpler because we only have to perform the query locally at each database and union the results. However, in a world where sites and databases are autonomous, it is very difficult to agree on and enforce global partitions. For example, the California DMV may want to keep a record of all violations that occurred in its state regardless of the origin of the driver's license. At the same time, the California DMV may not have complete records for California drivers because the other DMVs may not notify the California DMV of infractions occurring in their states involving California drivers.

We see this pattern occurring over and over: A phone company keeps three customer databases, managed by different organizations (billing, service & repairs, planning) and each with overlapping and differing information. A state library system has databases at

[1]The material presented in this chapter is based on ideas developed by the author with Yannis Papakonstantinou, Serge Abiteboul and Hector Garcia-Molina. Some of these ideas had been presented in the conference paper [81].

different locations, and there are multiple records for the same book, each with slightly different bibliographic information. There are many other fusion scenarios over the Web: the same products are described by different catalogs on the Web, the same books are offered by different on-line bookstores, and so on. In all these cases there is a strong need to query this autonomous information as if it were an integrated collection, to be able to discover opportunities (e.g., the same book is offered at the lowest price at one bookstore) or to have accurate information (e.g., we want to have all the facts about a customer to decide if we mail the customer promotional material).

Not only are fusion queries very important in the Web context, but they also introduce tough performance challenges. Even the best of query-execution plans will require a lot of work, since they have to check all the possible ways in which an entity can be found to match the selection conditions based on information from various sources. The amount of required work grows rapidly as the number of sources or the number of conditions to check grows. So, if we do not have good execution plans, it will be simply infeasible to answer nontrivial queries. Hence, the role of the query optimizer is very critical for fusion queries. Furthermore, the optimizer itself needs to be efficient as there are huge numbers of possible plans that can be considered.

Traditional query optimizers are not designed to handle fusion queries, and hence may produce poor plans. They typically assume that the number of sources is small, and rely on good fragmentation of data which as we have argued is not the case with data sources on the Web. Our goal in this chapter is to understand how fusion queries can be processed efficiently in the context of Web sources. To gain this knowledge, we proceed in four steps:

1. To make our task more manageable, we consider fusion queries that retrieve only the merge attribute (e.g., driver's license) of the matching entities. If additional information on the matching entities (e.g., their address) is needed, a "second phase" query could be issued. The assumption eliminates some performance factors (e.g., when should additional attributes be fetched), but still lets us study the basic types of fusion-query plans.

   Furthermore, the "two-phase" approach is used often in practice, so it is interesting in its own right. For instance, in a bibliographic-search scenario, one first identifies the documents that satisfy the criteria, and then fetches the documents, usually a few at a time. The main reason why searches are split this way is that the full records of the matching entities may be very large and are often stored on separate systems altogether. For example, a driver's name, address, and physical features may be stored separately from the violations information. Even when such is not the case, this two-phase processing may reduce cost because we do not pay the price of fetching full records until we know which ones are needed.

2. To understand the types of fusion-query plans, we first narrow down the space of plans to those we call *simple plans*. Simple plans are coordinated by the mediator as follows. The mediator can ask one or more sources to evaluate a condition, obtaining a set of values for the merge attribute. The mediator can also request a source to determine if a particular merge-attribute value has a matching element that satisfies a specified condition. By sending a stream of such requests, the mediator can emulate a *semijoin* operation on the set of merge-attribute values it computes. Finally, the mediator can combine the sets of merge-attribute values it obtains via union and intersection operations. For our sample query, one simple plan, call it $\mathcal{P}_1$, could be as follows: First, the mediator asks each source to give it all L values for drivers with $V = dui$ (see Figure 4.1). Then the mediator collects all these values into a set, and for each L value in this set it sends a request to all the sources, asking each source to check whether the passed L value has a matching $V = sp$. Those L values that have the matching V conditions will form the final answer.

   The class of simple plans is very large; it includes all the strategies that most real-world optimizers would currently develop for a fusion query, plus many other natural plans. Thus, even though we have narrowed down our search, we expect to find very good plans in this space, that are at least as efficient as those found by traditional optimizers.

3. The class of simple plans is still too large to be searched in a brute-force manner for an optimal plan. Fortunately, using theorems we develop in this chapter, we can constrain our search to a much smaller class of plans, those we call *semijoin-adaptive*. Intuitively, these are simple plans that work on the conditions of the query in some order, one condition at a time. (Plan $\mathcal{P}_1$ above is semijoin-adaptive, since it first considers one condition fully, and then moves on to the second one.) It turns out that semijoin-adaptive plans are very good under a general cost model that is well suited for Web context. In particular, if there are only two query conditions (as is common with most user queries) then the best semijoin-adaptive plan is also the best simple

plan. In this case the optimizer can perform a significantly smaller search over the space of semijoin-adaptive plans, and still find the best simple plan. Even if there are more conditions in the query, we argue that the best semijoin-adaptive plan provides an excellent heuristic.

4. Once we find the best semijoin-adaptive plan, we consider some variations of this plan that make it non-simple but that may improve performance further. In other words, as a "postoptimization" step, we consider a class of plans that is larger than simple plans, but we only perform a local optimization in the neighborhood of our best semijoin-adaptive plan. For example, one of the variations we consider is having the mediator use the set-difference operation. To illustrate, consider our sample plan $\mathcal{P}_1$. We leave the first part unchanged, obtaining the set $X_1 = \{J55, T80, T21\}$ of all L values that have $V = dui$ at some source. Now, instead of sending X_1 to all sources for a semijoin, we only send it to the first source. The first source (R_1) returns the subset of X_1 with $V = sp$, i.e., $Y_1 = \{T21\}$. Now the mediator knows that $T21$ is definitely an answer to the query, so when it goes to the second source, it does not have to send the full X_1 set; instead it sends $X_1 - Y_1$, reducing the amount of data that must be sent to the source. In Section 4.5, we discuss this and other postoptimization techniques.

In summary, the highlights of this chapter are as follows:

- We define and study the processing of fusion queries (Section 4.2).
- We define important classes of plans for fusion queries, including simple and semijoin-adaptive plans, and other subclasses of them (Section 4.2.5).
- Using a cost model that factors in both communication costs and query processing costs at the sources, we present an optimization algorithm for efficiently finding the optimal semijoin-adaptive plan (Section 4.3). The algorithm runs in time linear in the number of sources participating in the fusion, a significant benefit in scenarios involving the fusion of information from a large number of sources.
- We prove that in many commonly occurring situations, the optimal semijoin-adaptive plan is also the optimal plan under the broader class of simple plans (Section 4.4). This result lets us find the optimal simple plan with the same, efficient algorithm for semijoin-adaptive plans.

- We describe a set of postoptimization heuristics that can improve the performance of the optimal semijoin-adaptive plan further (Section 4.5).
- We present (Section 4.6) an evaluation of the fusion-query plans generated using the various techniques, quantify the gains achievable in some scenarios, and identify important trends.
- We discuss (Section 4.7) how the optimization strategies we have developed here can be incorporated into existing general-purpose optimizers in order to achieve efficient fusion-query processing.

4.2 Framework

In this section, we provide the framework for fusion-query optimization. First, we define the operations and data exported by sources. Second, we formally define fusion queries. Then we describe the class of simple plans, and our cost model. Finally, we identify some important subsets of simple plans.

4.2.1 The Sources

In our framework, each source has a *wrapper* [29] that exports a relation.[2] All the source relations have the same attributes, which include the merge attribute M. Attribute M identifies the real-world entity that the tuple refers to. Internally, each source can use a different model, but the wrapper maps it to the common view we are using. Note that we use a relational framework here only for simplicity. The algorithms we propose in this chapter can be extended in a straightforward way to other data models. Incidentally, our interest in the fusion problem emerged from the TSIMMIS project which uses a semistructured object model [13].

We assume here that all wrappers can indeed provide a common merge attribute M, such as the driver's license in our DMV example. In most cases of interest, there is such an attribute, but it may be represented in slightly different ways at each source. The differences can range from simple ones, e.g., one source uses hyphens between the state and the number in a license and another does not, to complex ones, e.g., customers are identified only by

[2]The wrapper can export other relations of course; here we focus on the one involved in the fusion query of interest.

name, and names can be misspelled. The most practical way to cope with these problems is to develop a unique *canonical form* for the merge attribute (and hopefully for the rest of the attributes too). Then the wrapper or the mediator uses mapping functions [54] to translate data from the source into the canonical form and translate queries with canonical values into queries that use source-specific values. Unfortunately, canonical forms are not always applicable (e.g., consider the case of misspellings.) However, to deal with a more tractable problem, we assume that canonical forms are available and used.

In our framework, wrappers support the following two types of operations:

- *Selection queries* denoted as $X := sq(c_i, R_j)$. This operation retrieves the set of items that satisfy c_i in source relation R_j (we use the term *item* to refer to a merge-attribute value). On the Web, for example, data sources typically offer limited capabilities, so conditions may be of the form "attribute equals constant". For a detailed discussion on condition processing at data sources, please refer to Chapter 2.

- *Semijoin queries* denoted as $X := sjq(c_i, R_j, Y)$. This operation computes the subset of Y items that satisfy c_i in R_j. There are sources that may not provide a semijoin operation directly. It is possible in such cases that the wrappers for such sources provide the semijoin capability. And in those cases where the wrappers also do not provide the semijoin operation, the mediator can "emulate" the semijoin by issuing individual selection queries, one for each value in Y. The cost of the emulated operation will be higher than if the wrapper supported the semijoin operation, and the higher cost will be taken into account in our model. Our assumption does imply that wrappers are at least able to handle selection conditions of the form c_i AND $M = m$, where m is a "passed binding" (from Y) and c_i is a condition in the fusion query.

4.2.2 Fusion Queries

We use U to refer to the union of all the source relations R_j. The general form of fusion queries is:

SELECT $u_1.M$
FROM $U\ u_1, \ldots, U\ u_m$
WHERE $u_1.M = \ldots = u_m.M$
AND c_1 AND ... AND c_m

where each condition c_i, $i = 1, \ldots, m$ involves only one u_i variable and the attributes of U, and is supported by the wrappers.[3] Note that the query only retrieves the merge attribute M, as discussed in Section 4.1.

4.2.3 Simple Plans

Under *simple plans*, mediators can issue selection and semijoin queries to the wrappers, and can themselves perform operations of the form $X := Y \ op \ Z$, where Y and Z are sets of items, and *op* is either a union ($\cup$) or an intersection ($\cap$). Notice that, in general, mediators would have to perform joins; however, in our framework for fusion queries, wrappers only return sets of items (unary relations), so joins are equivalent to intersections. A simple plan is thus a sequence of selection queries, semijoin queries, and local operations. Figures 4.2(a), 4.2(b) and 4.2(c), later on in this section, give examples of simple plans for a fusion query with 3 conditions and 2 sources.

Simple plans are quite general and can represent many ways to efficiently execute fusion queries. They allow the description of any plan obtained by standard algebraic optimization techniques such as pushing selection and projection operations to the sources. They also allow query rewriting using the distributivity of join and union, the commutativity and associativity of join and union, along with many techniques to reorder joins efficiently to process m-way joins, and the use of semijoin operations for efficient processing of joins in distributed environments. These are strategies that most optimizers typically use. Thus, if we are able to find the best simple plan, we believe we will have a plan that cannot be beaten by existing real-world optimizers. (There are, however, techniques not covered by simple plans, and some of them will be considered in Section 4.5.)

Simple plans can be employed in many contexts with a wide range of source capabilities. Even if some of the sources do not support semijoin queries, the mediator can emulate a semijoin query as a set of selection queries. Another approach to dealing with sources that do not support semijoin queries is to assign an infinite cost to that operation. All query plans that use the unsupported semijoin queries will be eliminated automatically by the optimizer (because of the infinite cost assigned to such plans).

[3] Actually, each u_i can refer to a different union view U_i which represents the union of all the source relations R_j that pertain to condition c_i. For simplicity of exposition we consider that all the U_is are the same. The entire discussion of this chapter and the techniques developed here are directly applicable to the case where the U_is may differ.

1)	$X_{11} := sq(c_1, R_1)$	1)	$X_{11} := sq(c_1, R_1)$	1)	$X_{11} := sq(c_1, R_1)$
2)	$X_{12} := sq(c_1, R_2)$	2)	$X_{12} := sq(c_1, R_2)$	2)	$X_{12} := sq(c_1, R_2)$
3)	$X_1 := X_{11} \cup X_{12}$	3)	$X_1 := X_{11} \cup X_{12}$	3)	$X_1 := X_{11} \cup X_{12}$
4)	$X_{21} := sq(c_2, R_1)$	4)	$X_{21} := sjq(c_2, R_1, X_1)$	4)	$X_{21} := sjq(c_2, R_1, X_1)$
5)	$X_{22} := sq(c_2, R_2)$	5)	$X_{22} := sjq(c_2, R_2, X_1)$	5)	$X_{22} := sq(c_2, R_2)$
6)	$X_2 := X_{21} \cup X_{22}$	6)	$X_2 := X_{21} \cup X_{22}$	6)	$X_2 := X_{21} \cup X_{22}$
7)	$X_2 := X_2 \cap X_1$	7)	$X_{31} := sq(c_3, R_1)$	7)	$X_2 := X_2 \cap X_1$
8)	$X_{31} := sq(c_3, R_1)$	8)	$X_{32} := sq(c_3, R_2)$	8)	$X_{31} := sq(c_3, R_1)$
9)	$X_{32} := sq(c_3, R_2)$	9)	$X_3 := X_{31} \cup X_{32}$	9)	$X_{32} := sq(c_3, R_2)$
10)	$X_3 := X_{31} \cup X_{32}$	10)	$X_3 := X_2 \cap X_3$	10)	$X_3 := X_{31} \cup X_{32}$
11)	$X_3 := X_3 \cap X_2$			11)	$X_3 := X_2 \cap X_3$
(a)	A Filter Plan.	(b)	A Semijoin Plan.	(c)	A Semijoin-adaptive Plan.

Figure 4.2: Three Simple Plans.

4.2.4 Cost Model

In our domain of interest, the Web data sources, the most time consuming task is sending queries to the sources and receiving answers from them. Thus, as in Chapters 2 and 3, we adopt a model that emphasizes these costs and neglects the cost of local processing at the mediator. In particular,

- Each $sq(c_i, R_j)$ and each $sjq(c_i, R_j, X)$ operation has a non-negative cost, given by $f_j + v_j * (\textit{size of the query result})$.
- If X, Y and Z are sets of items with $X = Y \cup Z$, the cost of $sjq(c_i, R_j, X)$ is at most the sum of the costs of $sjq(c_i, R_j, Y)$ and $sjq(c_i, R_j, Z)$ for any c_i and R_j. In other words, there is no benefit in splitting a semijoin set X into semijoin sets Y and Z.
- The cost of local mediator operations, $\cup$ and $\cap$, is negligible.
- The cost of a query plan is the sum of the costs of the constituent $sq(c_i, R_j)$ and $sjq(c_i, R_j, X)$ operations. Thus, our focus is on total work involved in query execution, not on response time. (We briefly discuss response time issues in Section 4.8.)

4.2.5 Important Classes of Simple Plans

To conclude this section, we define a few important classes of simple plans for fusion queries.

1. *Filter plans*: Filter plans are simple plans that use only selection queries and local operations at the mediator. Many traditional distributed-query optimizers do not use semijoin operations [22, 63]. Such optimizers generate filter plans for fusion queries.

   Figure 4.2(a) shows an example filter plan for a fusion query with conditions c_1, c_2, and c_3 and sources R_1 and R_2. In this plan, the mediator pushes each condition to each source (six selection queries), and computes the final answer from the corresponding item sets. In general, every filter plan issues mn selection queries, where m is the number of conditions and n is the number of sources, and performs local computations at the mediator. Of course, the order of these operations can be different from that in Figure 4.2(a).

2. *Semijoin plans*: These are simple plans that employ semijoin queries in a restricted fashion. A particular semijoin plan is determined first by an ordering, say, $c_1, ..., c_m$ of the query conditions. The set X_1 of items satisfying c_1 at some source is first retrieved by issuing selection queries, one for each of the n sources. Next, the plan can either evaluate the second condition in a similar fashion, or it can evaluate it by semijoin queries using X_1 as semijoin set. In either case, the plan computes X_2, the set of items that satisfy c_1 at one source and c_2 at another (possibly the same) source. The process continues in a similar fashion for the rest of the conditions. In general, for a given ordering of the conditions, a particular semijoin plan is specified by deciding for each condition c_i (i in $[2..m]$), whether to evaluate c_i by selection queries or by semijoin queries using as semijoin set the set of items satisfying $c_1 \wedge ... \wedge c_{i-1}$.

   Figure 4.2(b) illustrates a semijoin plan for the same fusion query used in Figure 4.2(a). The second condition is evaluated by semijoin queries, and the others by selection queries. Observe that the first condition in a semijoin plan is always evaluated by selection queries, as there is no semijoin set to start with.

   Semijoin plans can be more efficient than filter plans. For instance, in Figure 4.2(b), the source with R_1 only returns a fraction of the items satisfying c_2, while the equivalent filter plan would have fetched all items in R_1 satisfying c_2. However, we can make semijoin plans even more effective by allowing them more flexibility. In particular, notice that for a given condition, semijoin plans either send selection queries to all sources or send semijoin queries to all sources. Doing so may be inefficient in an environment where the sources have widely different characteristics. For example, if the second source does not directly support semijoins (i.e., semijoins have to be emulated in an expensive manner) it may not be beneficial to process c_2 at the second source using semijoins. The class of plans described next has the ability to adapt to the characteristics of the sources.

3. *Semijoin-adaptive plans*: Like semijoin plans, these plans process one condition at a time, in some order $c_1, ..., c_m$. However, for each condition in $[2..m]$ and each source, the plan can choose independently between a selection query or a semijoin query. Figure 4.2(c) illustrates a semijoin-adaptive plan for the same sample query of the previous figures. As we can see, the plan processes c_2 at R_1 by issuing a semijoin query, and at R_2 by the use of a selection query. Thus, the plan can use the best strategy at each source.

It is important to note that semijoin plans are a special case of semijoin-adaptive plans, which in turn, are a special case of simple plans, as illustrated by Figure 4.3. Filter plans are *not* a subset of semijoin-adaptive plans, since they are not restricted to compute the sets X_1, $X_2, ...$ used in the definition above. (They are of course not a superset since they do not use semijoins.) However, under our cost model, all filter plans have the same cost, i.e., the cost of performing the mn selection queries. Neither the order in which the selection queries are performed in a filter plan nor the mediator operations executed changes this cost. Therefore, for query-optimization purposes, we can pick a single *reference* filter plan within the space of filter plans, like the one in Figure 4.2(a), and ignore the rest. Since this reference filter plan also lies in the space of semijoin plans, we can *conceptually* think of filter plans as a subset of semijoin plans.

In Section 4.4, we will prove that in some cases the optimal semijoin-adaptive plan is also optimal within the space of all simple plans. Furthermore, in Section 4.6, we compare the performance of our plans and demonstrate that semijoin-adaptive plans have a significant performance advantage over filter and semijoin plans. However, before that, we discuss how to find optimal plans within the plan spaces of Figure 4.3.

4.3 Finding Optimal Simple Plans

In this section we present three algorithms — FILTER, SJ, and SJA — that compute the best filter, semijoin, and semijoin-adaptive plans respectively. These algorithms are very

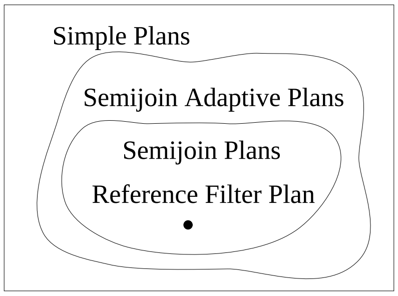

Figure 4.3: Relationship between the Subsets of Simple Plans.

efficient, as they run in time linear in the number of sources participating in the fusion query.

The algorithms use the following:

1. A function $sq\_cost(c_i, R_j)$ to estimate the cost of the selection query $sq(c_i, R_j)$.

2. A function $sjq\_cost(c_i, R_j, X)$ to estimate the cost of the semijoin query $sjq(c_i, R_j, X)$.

These functions can use whatever information is available at query-optimization time, e.g., the expected size of data sets at the sources, the selectivities of the conditions at the various sources, and the capabilities of the sources (i.e., whether they can support semijoin operations directly or not). (For an illustration of particular cost estimates please see Section 4.6.) In analyzing the complexity of the various algorithms presented in this section, we assume that *sq_cost* and *sjq_cost* take constant time per invocation.

The FILTER algorithm: FILTER directly outputs the single reference filter plan (see Section 4.2.5). Hence FILTER does not spend any time searching for the optimal plan. Its running time is proportional to the number of steps required for producing the reference filter plan which, in turn, is proportional to the number of statements in the reference filter plan. Hence, the complexity of FILTER is $O(mn)$, where m is the number of conditions and n is the number of sources.

The SJ algorithm: SJ (see Figure 4.4) generates all possible $m!$ orderings of the conditions. For each one of them, it generates the best *Plan* with respect to this ordering, estimates its cost, and eventually selects as *Optimal Plan* the one with the least cost among all orderings.

Algorithm 4.3.1 *SJ*
Input: Conditions $c_1, \ldots, c_m$; Sources $R_1, \ldots, R_n$; Cost functions sq_cost and sjq_cost
Output: An optimal semijoin plan
Method:

$M \leftarrow infinity$
For every ordering $[c_{o_1}, \ldots, c_{o_m}]$ of the conditions
  $P \leftarrow [X_{11} := sq(c_{o_1}, R_1), \ldots, X_{1n} := sq(c_{o_1}, R_n)]$
  $P \leftarrow P \cdot [X_1 := \cup_{j=1,\ldots,n} X_{1j}]$
  $K \leftarrow \sum_{j=1,\ldots,n} sq\_cost(c_{o_1}, R_j)$
  For $i = 2, \ldots, m$
    $K_1 \leftarrow \sum_{j=1,\ldots,n} sq\_cost(c_{o_i}, R_j)$
    $K_2 \leftarrow \sum_{j=1,\ldots,n} sjq\_cost(c_{o_i}, R_j, X_{i-1})$
    If $(K_1 < K_2)$
      $P \leftarrow P \cdot [X_{i1} := sq(c_{o_i}, R_1), \ldots, X_{in} := sq(c_{o_i}, R_n)]$
      $P \leftarrow P \cdot [X_i := X_{i-1} \cap (\cup_{j=1,\ldots,n} X_{ij})]$
      $K \leftarrow K + K_1$
    Else
      $P \leftarrow P \cdot [X_{i1} := sjq(c_{o_i}, R_1, X_{i-1}), \ldots, X_{in} := sjq(c_{o_i}, R_n, X_{i-1})]$
      $P \leftarrow P \cdot [X_i := \cup_{j=1,\ldots,n} X_{ij}]$
      $K \leftarrow K + K_2$
  If $(K < M)$
    $OPT \leftarrow P$
    $M \leftarrow K$
Return(OPT) $\diamond$

Figure 4.4: The SJ Algorithm.

Algorithm 4.3.2 *SJA*
Input: Conditions $c_1, \ldots, c_m$; Sources $R_1, \ldots, R_n$; Cost functions sq_cost and sjq_cost
Sources $R_1, \ldots, R_n$
Cost functions sq_cost and sjq_cost
Output: An optimal semijoin-adaptive plan
Method:

$M \leftarrow infinity$
For every ordering $[c_{o_1}, \ldots, c_{o_m}]$ of the conditions
  $P \leftarrow [X_{11} := sq(c_{o_1}, R_1), \ldots, X_{1n} := sq(c_{o_1}, R_n)]$
  $P \leftarrow P \cdot [X_1 := \cup_{j=1,\ldots,n} X_{1j}]$
  $K \leftarrow \sum_{j=1,\ldots,n} sq\_cost(c_{o_1}, R_j)$
  For $i = 2, \ldots, m$
    For $j = 1, \ldots, n$
      If $sq\_cost(c_{o_i}, R_j) < sjq\_cost(c_{o_i}, R_j, X_{i-1})$
        $P \leftarrow P \cdot [X_{ij} := sq(c_{o_i}, R_j)]$
        $K \leftarrow K + sq\_cost(c_{o_i}, R_j)$
      Else
        $P \leftarrow P \cdot [X_{ij} := sjq(c_{o_i}, R_j, X_{i-1})]$
        $K \leftarrow K + sjq\_cost(c_{o_i}, R_j, X_{i-1})$
    $P \leftarrow P \cdot [X_i := X_{i-1} \cap (\cup_{j=1,\ldots,n} X_{ij})]$
  If $(K < M)$
    $OPT \leftarrow P$
    $M \leftarrow K$
Return(OPT) $\diamond$

Figure 4.5: The SJA Algorithm.

The best *Plan* with respect to a specific ordering $[c_{o_1}, \ldots, c_{o_m}]$ starts with a sequence of operations that evaluate c_{o_1} using selection queries. Then SJ goes over each one of the $m-1$ conditions c_{o_i}, $i = 2, \ldots, m$ and decides whether c_{o_i} is evaluated by semijoin or selection queries. In particular, SJ sums up and compares the cost of the n selection queries with the cost of the n semijoin queries. The *Plan* and its cost are appropriately updated in each round. The complexity of SJ is $O((m!)mn)$ because the outer loop iterates $m!$ times, the inner loop iterates $m-1$ times, and the operations inside the inner loop are of complexity $O(n)$.

The SJA algorithm: SJA (see Figure 4.5) differs from the SJ algorithm in that it makes a separate decision between selection and semijoin query for each condition at each source. In particular, SJA includes the "source loop" of Figure 4.5 where, for a given condition c_{o_i}, it decides, for each source R_j, whether the processing of c_{o_i} at R_j will be done with a semijoin query or a selection query.

It is easy to see that SJA's complexity is also $O((m!)mn)$ since the source loop iterates n times and the operations inside it cost $O(1)$. The complexity of SJA is similar to that of SJ, despite the fact that the space of semijoin-adaptive plans is much larger than the space of semijoin plans (there are $O((m!)2^{m-2})$ semijoin plans, for a given query with m conditions, assuming we do not consider semijoin plans that are equivalent with respect to our cost model; and there are $O((m!)2^{n(m-2)})$ semijoin-adaptive plans). Moreover, the optimal semijoin-adaptive plan is always at least as good as, and often much better than (see Section 4.6), the optimal semijoin plan. So, SJA is preferable to SJ.

The fact that the algorithms presented in this section run in time linear in the number of sources is very important when we deal with a large number of sources as is the case with integrating Web data sources. The running times of SJ and SJA are exponential in the number of conditions. In most realistic scenarios, the exponential running time is acceptable since the number of conditions (unlike the number of sources) is small. However, if the number of conditions is large then we can use greedy versions of SJ and SJA that work on the conditions in ascending order of *overall selectivity* instead of trying all orderings. We define the overall selectivity of a condition c_i as the total number of items at all sources that satisfy c_i. Because they consider only one condition ordering instead of $m!$, the greedy versions run in $O(mn)$ time.[4]

4.4 Optimality of Semijoin-Adaptive Plans

In this section, we investigate the optimality of semijoin-adaptive plans within the space of all simple plans. We demonstrate that for queries with less than three conditions, for each simple plan $\mathcal{P}$, there is a semijoin-adaptive plan that performs at least as well as $\mathcal{P}$ on every database. That is, the optimal simple plan falls within the subspace of semijoin-adaptive plans. Later we show that when there are three or more conditions the optimal simple plan may be a non-semijoin-adaptive plan. However, we argue that this non-optimality of

[4]The actual run time complexity is $O(mn + m \log m)$, but $logm$ is usually much smaller than n.

semijoin-adaptive plans only occurs in certain "rare" scenarios involving condition correlations.

Before we proceed, we define some notation. Let $c_1, ..., c_m$ $(m \geq 1)$ be an ordering of the conditions of the fusion query. Let $X_1, ..., X_m, X_{11}, ...X_{mn}$ be temporary sets (unary relations). For each $i \in [1..m]$ and $j \in [1..n]$, let s_{ij} be defined by:

$$\begin{aligned} s_{1j}: &\quad X_{1j} := sq(c_1, R_j) \\ s_{ij}: &\quad X_{ij} := sq(c_i, R_j) \quad \text{or} \quad X_{ij} := sjq(c_i, R_j, X_{i-1}) \quad \text{if } i > 1 \end{aligned}$$

and let s'_{ij} be the statement $X_i := X_i \cup X_{ij}$. With this notation we can write a semijoin-adaptive plan as $\mathcal{P} = \mathcal{P}_1; ...\mathcal{P}_m$ where

$$\begin{aligned} \mathcal{P}_1 &= s_{11}; ...; s_{1n}; s'_{11}; ...; s'_{1n} \\ \mathcal{P}_i &= s_{i1}; ...; s_{in}; s'_{i1}; ...; s'_{in}\ X_i := X_{i-1} \cap X_i; \quad \text{if } i > 1. \end{aligned}$$

We also need the auxiliary notion of i-semijoin-adaptive plans. A plan $\mathcal{P}_1; ...; \mathcal{P}_i; \mathcal{P}'$ is *i-semijoin-adaptive* if (i) $\mathcal{P}_1; ...; \mathcal{P}_i$ is semijoin-adaptive for conditions $c_1, ..., c_i$ and (ii) $\mathcal{P}'$ contains no selection or semijoin query in which any condition in $c_1, ..., c_i$ occurs. Note that if m is the number of conditions, an m-semijoin-adaptive plan is semijoin-adaptive.

We first prove an *initialization* lemma that will lead us to our result for fusion queries with fewer than three conditions (Theorem 4.4.1).

Lemma 4.4.1 *Let $\mathcal{P}$ be a simple plan computing a query Q. There exists a 1-semijoin-adaptive plan computing Q and costing never more than $\mathcal{P}$ on any database.* □

Proof: First, suppose that there exists some i such that, for each source j, $sq(c_i, R_j)$ occurs in $\mathcal{P}$. Then, it is easy to transform $\mathcal{P}$ into a 1-semijoin-adaptive plan: add in some preamble to $\mathcal{P}$, $X_{ij} := sq(c_i, R_j)$ for each j, and eliminate all selection or semijoin queries for condition c_i using the X_{ij}'s. The resulting plan will be 1-semijoin-adaptive and will perform at least as well as $\mathcal{P}$ on every database.

Now suppose that no such i exists for $\mathcal{P}$. Then, for each i, there exists $j(i)$ such that $sq(c_i, R_{j(i)})$ does not occur in $\mathcal{P}$. We then construct a database I as follows: For each i, relation $R_{j(i)}$ has a tuple that satisfies c_i and has a merge attribute value of γ, for some constant γ. Besides these, no other tuples exist in the relations. Then, the merge attribute γ satisfies all the conditions of the query Q. Therefore, Q has a nonempty result on I. However, $\mathcal{P}$ returns an empty result for I, a contradiction since $\mathcal{P}$ was supposed to compute Q. ∎

Theorem 4.4.1 *Let Q be a query with fewer than three conditions and $\mathcal{P}$ a simple plan computing Q. Then, there exists a semijoin-adaptive plan computing Q and costing never more than $\mathcal{P}$ on any database.*

Proof: By Lemma 4.4.1, there exists a 1-semijoin-adaptive plan $\mathcal{P}_1;\mathcal{P}'$ computing Q and costing never more than $\mathcal{P}$ on any database. If Q has only one condition, the result is trivially obtained because $\mathcal{P}'$ would be empty. Suppose it has two conditions, c_1 in $\mathcal{P}_1$ and c_2 in $\mathcal{P}'$. Now, this plan is "almost" semijoin-adaptive. Indeed, $\mathcal{P}_1$ is a prefix of a semijoin-adaptive plan and after $\mathcal{P}_1$, we know the set X_1 of items satisfying c_1. Now, consider each source j in turn. One of the following cases arises:

1. $sq(c_2, R_j)$ occurs in $\mathcal{P}'$. Then we can move this selection early and remove all other queries with c_2 to R_j.
2. $sjq(c_2, R_j, X_1)$ occurs in $\mathcal{P}'$. Similarly, we can move this semijoin query early and remove all other queries with c_2 to R_j.
3. Neither of the above two cases holds. Then, $\mathcal{P}$ can get the proper subset of X_1 satisfying c_2 at Source j only by using several semijoin queries. But each item in X_1 would have to be tried. So, we can not increase the cost by introducing $sjq(c_2, R_j, X_1)$ early and removing all queries with c_2 to R_j.

Thus, we can obtain a semijoin-adaptive plan computing Q and costing never more than $\mathcal{P}_1;\mathcal{P}'$ on any database. ∎

Given that Theorem 4.4.1 asserts the optimality of semijoin-adaptive plans for up to two conditions, it is natural to wonder if semijoin-adaptive plans are optimal for an arbitrary number of conditions. It turns out that it is not the case.

Lemma 4.4.2 *There is a query Q with 3 conditions (and 2 sources), a non-semijoin-adaptive plan $\mathcal{P}$, a database I, such that every semijoin-adaptive plan computing Q costs strictly more than $\mathcal{P}$ on I.* □

Proof: Let R_1, R_2 be two sources and c_1, c_2, c_3 be three conditions. We describe a database I and then a non-semijoin-adaptive plan $\mathcal{P}$. First, $I(R_1)$ and $I(R_2)$ have 100 million items each, and c_1 has a selectivity of 0.001. For items from R_1 satisfying c_1, c_2 has selectivity 0.1 and c_3 has selectivity 0.9. For items from R_2 satisfying c_1, c_2 has selectivity 0.9 and c_3 has selectivity 0.1. There are no items that satisfy c_1 in both R_1 and R_2. Finally, c_2 and c_3 have overall selectivity of 0.95. This data distribution suggests the non-semijoin-adaptive plan $\mathcal{P}$: (1) compute the items satisfying c_1 using two selection queries; (2) for those items found in R_1 satisfying c_1, consider c_2, then c_3 (by semijoins); (3) for those items found in R_2 satisfying c_1, consider c_3, then c_2 (again by semijoins).

On database I, $\mathcal{P}$ strictly outperforms any semijoin-adaptive plan. Thus, we have shown that there is a query with three conditions on two sources for which the best semijoin-adaptive plans is not the optimal simple plan. ∎

The key feature of the plan in Lemma 4.4.2 is that after evaluating c_1 at both R_1 and R_2, the next steps do not simply use X_1, the complete set of items satisfying c_1. Instead, the items satisfying c_1 at R_1 are treated differently from those satisfying c_1 at R_2. The only reason why the different treatment pays off is that we have strong "intersource" dependencies, e.g., the items satisfying c_1 at R_1 have different selectivities for c_2 than the other items satisfying c_1 in general. Given a set of conditions $c_1, ..., c_i$, we call the set of items that satisfy the set of conditions over all the sources the *complete set* (for $c_1, ..., c_i$). A set of items that is not complete is said to be *incomplete*. In our example, the plan uses incomplete sets (e.g., the set of items satisfying c_1 at R_1 only and the set of items satisfying c_1 at R_2 only) as inputs to its semijoin queries. In contrast, semijoin-adaptive plans use only complete sets of items as inputs to their semijoin queries.

We now argue informally that if conditions are independent, plans that use incomplete sets will do no better than plans that use complete sets. Intuitively, there is no benefit to treating the items that satisfy a condition at one source differently from those that satisfy the same condition at a different source. If conditions are independent, the total amount of work cannot be increased if we simply combine these incomplete sets into a complete one, and adjust our plan accordingly.

Furthermore, even if there are some condition correlations, plans that use complete sets may still be as good as any. For example, suppose we only have *intercondition* dependencies. For instance, let the fact that an item satisfies condition c_1 make it more likely that it will satisfy c_2, independent of the sources involved. Under this kind of correlation, there is again no advantage to using incomplete sets in the plan. Thus, we still expect plans that use complete sets to be as good as any. Of course, with *intersource* dependencies (e.g., an item satisfying c_1 at R_1 makes it much more likely that the same item will satisfy c_2 at R_2) it does pay to use incomplete sets, as Lemma 4.4.2 illustrated.

Our discussion leads us to conjecture that for each query Q and each probabilistic data distribution with no intersource dependencies, there exists a semijoin-adaptive plan $\mathcal{P}$ computing Q that performs *on average* at least as well as any other simple plan.

It should be emphasized that even when intersource dependencies exist, one might still want to consider only semijoin-adaptive plans for a number of reasons: (i) information about the correlations is unlikely to be available in contexts like the Web; (ii) the search space to consider (when not limited to semijoin-adaptive plans) becomes prohibitively large; and (iii) the gains will most often be relatively small even with very distorted data distributions.

4.5 Postoptimization

In this section, we consider a number of postoptimization techniques that can improve the plans generated by the SJA algorithm. First, we describe two of them that we have implemented in an algorithm named SJA+. Then, we briefly discuss the SJA+ algorithm. The section ends with a description of a set of other postoptimization techniques.

Loading entire sources. Instead of sending a set of queries to a source, the mediator may consider issuing a single query to load the entire source contents and then using this result to evaluate all the queries for that source. This technique can be advantageous in fusion queries involving small-sized source databases or large number of conditions.

To illustrate, consider the two queries on R_3 in $\mathcal{P}_1$ (Steps 3 and 8 in Figure 4.6(a)). Let the cost of loading the entire contents of R_3 be lower than the cost of issuing the two queries on R_3. Plan $\mathcal{P}_{2a}$ in Figure 4.6(b) is the result of postoptimizing $\mathcal{P}_1$ by loading R_3 and replacing the two queries of $\mathcal{P}_1$ on R_3 by local computation at the mediator. Note that $lq(R_j)$ is a new operation primitive used to represent the operation of loading the entire relation R_j. Also observe that we use $sq(c_i, Y)$ to stand for the local application of the condition c_i on a set Y of items.[5]

Using the difference operation. A significant portion of the cost of semijoin queries to sources is directly proportional to the size of the semijoin sets of items. One way to reduce the size of the semijoin sets is to use the set *difference* operation as part of the local computation at the mediator. The consequent improvements are significant in situations involving sources that do not support semijoins directly and the semijoin operation has to be emulated.

In Section 4.1, we gave a simple example of postoptimization using the difference operator. Here we give a second example, now couched in our notation. Consider again plan $\mathcal{P}_1$

[5]Strictly speaking, Y is not a set of items because it may also include values for non-merge attributes on which the condition has to be applied.

1) $X_{11} := sq(c_1, R_1)$
2) $X_{12} := sq(c_1, R_2)$
3) $X_{13} := sq(c_1, R_3)$
4) $X_1 := X_{11} \cup X_{12}$
5) $X_1 := X_1 \cup X_{13}$
6) $X_{21} := sq(c_2, R_1)$
7) $X_{22} := sjq(c_2, R_2, X_1)$
8) $X_{23} := sq(c_2, R_3)$
9) $X_2 := X_{21} \cup X_{22}$
10) $X_2 := X_2 \cup X_{23}$
11) $X_2 := X_2 \cap X_1$

(a) Phase 1: $\mathcal{P}_1$.

1) $X_{11} := sq(c_1, R_1)$
2) $X_{12} := sq(c_1, R_2)$
3) $Y := lq(R_3)$
4) $X_{13} := sq(c_1, Y)$
5) $X_1 := X_{11} \cup X_{12}$
6) $X_1 := X_1 \cup X_{13}$
7) $X_{21} := sq(c_2, R_1)$
8) $X_{22} := sjq(c_2, R_2, X_1)$
9) $X_{23} := sq(c_2, Y)$
10) $X_2 := X_{21} \cup X_{22}$
11) $X_2 := X_2 \cup X_{23}$
12) $X_2 := X_2 \cap X_1$

(b) Loading Sources: $\mathcal{P}_{2a}$.

1) $X_{11} := sq(c_1, R_1)$
2) $X_{12} := sq(c_1, R_2)$
3) $X_{13} := sq(c_1, R_3)$
4) $X_1 := X_{11} \cup X_{12}$
5) $X_1 := X_1 \cup X_{13}$
6) $X_{21} := sq(c_2, R_1)$
7) $Z_1 := X_1 - X_{21}$
8) $X_{22} := sjq(c_2, R_2, Z_1)$
9) $X_{23} := sq(c_2, R_3)$
10) $X_2 := X_{21} \cup X_{22}$
11) $X_2 := X_2 \cup X_{23}$
12) $X_2 := X_2 \cap X_1$

(c) Using Difference: $\mathcal{P}_{2b}$.

1) $X_{11} := sq(c_1, R_1)$
2) $X_{12} := sq(c_1, R_2)$
3) $Y := lq(R_3)$
4) $X_{13} := sq(c_1, Y)$
5) $X_1 := X_{11} \cup X_{12}$
6) $X_1 := X_1 \cup X_{13}$
7) $X_{21} := sq(c_2, R_1)$
8) $Z_1 := X_1 - X_{21}$
9) $X_{22} := sjq(c_2, R_2, Z_1)$
10) $X_{23} := sq(c_2, Y)$
11) $X_2 := X_{21} \cup X_{22}$
12) $X_2 := X_2 \cup X_{23}$
13) $X_2 := X_2 \cap X_1$

(d) SJA+ Choice: $\mathcal{P}_2$.

Figure 4.6: Postoptimization.

of Figure 4.6(a). In Step 7, $\mathcal{P}_1$ issues a semijoin query. At the end of Step 5, X_1 contains all the items that satisfy c_1. In Step 6, X_{21} collects all the items of relation R_1 that satisfy condition c_2. From X_1 and X_{21}, we can find the set of items that have already satisfied c_1 and c_2. These items need not be sent to the source in Step 7, to ascertain the satisfaction of condition c_2 in source relation R_2. Thus, there is no need to send the entire set X_1 as the semijoin input in Step 7. Instead, we can just send $X_1 - X_{21}$. Figure 4.6(c) shows the resulting plan.

4.5.1 SJA+

The SJA+ algorithm incorporates the above two postoptimization techniques as follows. First, it mimics SJA to obtain the best semijoin-adaptive plan for the given query. Then, it uses the difference operation to prune the semijoin sets, in all the semijoin queries as described above. Finally, it considers the option of loading entire source contents to further improve the plan. Figure 4.6(d) shows a plan that may be obtained by SJA+, assuming that the plan of Figure 4.6(a) is obtained by SJA for the same fusion query.

The time complexity of SJA+ is $O((m!)mn + mn)$. The $(m!)mn$ term is the cost of SJA. The second term mn is for postoptimization, computed as follows. The postoptimization phase in SJA+ considers the semijoin queries of $(m-1)$ conditions to be improved upon, by using the difference operation. For each set of semijoin queries corresponding to a condition, SJA+ spends $O(n)$ time reducing the semijoin sets and modifying the semijoin queries appropriately. Thus, postoptimization using the difference operation takes $O(mn)$ time. Then, for each of the n sources, SJA+ takes $O(m)$ time to decide on replacing all its queries by an lq operation and local computation at the mediator. The modification of the plan also takes $O(m)$ per source, because SJA+ will replace m steps by $1 + 2m$ steps (the first to load the entire source and the rest for local computation). Thus, the total postoptimization cost is $O(mn)$. Note that SJA+ has the same order of complexity as SJA. In particular, the postoptimization phase of SJA+ is very efficient.

We note that the postoptimization phase of SJA+ uses operations that are not allowed in simple plans as defined in Section 4.2. In this sense, SJA+ yields plans outside the space of simple plans. One could have considered the more general class of plans up front, and systematically searched for optimality within that space. We decided not to follow that approach because of the very large number of plans that must be then considered to find an optimal plan. For instance, the extended SJA algorithm would be exponential in n.

Given that n is usually large in the application domains of interest to us, such algorithms are infeasible.

4.5.2 Other Postoptimization Techniques

Here, we briefly describe a few other techniques that can be employed in postoptimizing the plans generated by our algorithms of Section 4.3.

Loading relevant portions of sources. This technique is similar to loading the entire sources. It may be wasteful to load a large portion of a source that does not contribute to the result of the fusion query because it does not satisfy any of the specified conditions. In order to avoid this waste, one may first gather all the items that satisfy the first (perhaps, the most selective) condition. Next, this set of items is sent to each source with a request to return all the relevant attributes of the items. The relevant attributes are those that appear in the conditions specified in the fusion query. Then, the mediator evaluates the rest of the conditions locally. This technique can be adopted at any stage in the semijoin-adaptive plan, where a set of items satisfying a subset of conditions has been computed. The case where this subset of conditions is empty corresponds to the technique of loading entire sources discussed earlier.

Issuing disjunctive queries to sources. In this technique, the mediator sends a disjunction of some conditions, in a single query to a source. The source returns all the items that satisfy any of the conditions at the source. The mediator then performs local computations on this result instead of sending any selection or semijoin queries to the source. In order to facilitate the local processing at the mediator, the source needs to return additional information along with the result set of items in response to a disjunctive query. This information may be in the form of tags attached to each item in the result, indicating the list of conditions satisfied by the item. The source may instead return non-merge attributes also in the result, that will aid the mediator in locally applying the conditions, one at a time, on the disjunctive-query result. This technique requires that sources support disjunctive queries, and provide appropriate additional information to the mediator.

Using semantic information. We do not mean by semantic information here, fragmentation information like in a well-designed distributed-database system. We mean here semantic information that can be more realistically encountered in the Web setting such as key dependencies, or the absence of some attributes in some sources in the case of heterogeneous schemas. It is easy to take advantage of such knowledge. In the example of the fusion query on DMV sources, if a source does not record traffic violations, it is not necessary to ask that source the set of drivers with a *dui* violation. This technique leads to the elimination of some of the source queries from the plan and thus can reduce its cost.

Caching at the mediator the sets of items from sources. Suppose that the system maintains at the mediator the set of items from each source in relations $I_1, ..., I_n$. Note that it may be reasonable to cache this kind of information at a mediator since it does not depend on any specific fusion query. It may provide speed-up for instance by allowing us to replace a semijoin with some X at source j by a semijoin with $X \cap I_j$ since only items that the source knows of may participate successfully in the semijoin. The consequent reduction in the size of the semijoin sets in semijoin queries will yield improved plans.

Using bit vectors. In this technique, the first condition is sent to each source. Each source returns a bit vector B_j describing the items that passed the condition at that source. The bit vectors are merged at the mediator into a vector B which is sent to each source, along with the list of m conditions to satisfy. Each source then sends back a $k \times m$ bit matrix, where k is the length of B, describing the satisfaction of the m conditions for each item in B. All the rest of the computation can be carried out locally by the mediator. Essentially, this technique is using the ability to process source queries with disjunctive condition specification, along with cooperation among the sources regarding the mapping of sets of data items to bit vectors. It is to be noted that such a technique is rarely feasible in the context of Web data sources since a majority of the data sources do not support such sophisticated queries and operations. However, we can at least consider the idea of using compact representations to pass sets of items between the mediator and the sources as a postoptimization technique to improve upon the best plan generated by SJA.

4.6 Performance Evaluation

In this section we study the performance of our optimization algorithms. We wish to address the following questions:

- Knowing the space of plans each algorithm considers, we obviously expect SJA plans to be superior to SJ plans, and in turn superior to FILTER plans. However, the important question here is: do the plans of SJA far outperform the plans of SJ and FILTER? We ask a similar question regarding the performance gains of SJA+ over SJA: what is the extent to which SJA+ improves upon SJA?

- Correlations of query conditions can affect results in two ways, depending on whether the algorithm is aware of the correlations or not. First, if the algorithm is unaware of correlations (it falsely assumes independence, but correlations exist), it may actually yield a plan that is not even optimal in its target class. In this scenario, could it be that, for instance, SJA gives us a *worse* plan than SJ or FILTER? Second, if SJA takes into consideration the correlation information and yields the best semijoin-adaptive plan, this plan may not be the optimal simple plan (see Section 4.4). How far from the optimal will the plan found by SJA be?

Obviously it is very difficult to answer these questions in absolute terms since there are so many parameters to consider (e.g., source sizes, number of conditions, correlations). Instead, we have considered some "representative scenarios" to gain some insight into the expected answers. We report here on our study of these representation scenarios, and we summarize our empirical answers.

Cost functions. To evaluate the costs of our sample plans we use cost functions that compute the costs of selection queries and semijoin queries. To keep the model simple, we assume uniform cost-function expressions across all sources. That is, all selection queries use the same expression to compute their costs, while all semijoin queries use a second expression to compute their costs. For each selection query, we charge $q + r + o * \alpha$, where q is the fixed cost for a query, r is the fixed cost of an answer, α is the number of returned items, and o is a proportionality constant. For each semijoin query, we charge $q + i * \beta + r + o * \alpha$ where q, r, o and α are same as before, β is the number of items sent in the semijoin set and i is a proportionality constant. In particular, we choose $q = r = 1$ and $i = o = 1/100$. (If we are considering communication costs, for example, 100 result items or 100 semijoin items can be transmitted for the same cost as the header of a query or a result descriptor.)

We compute the sizes of the results of queries from the information about the sizes of the sources and the selectivities of the conditions at the sources. That is, the size of the result of a selection query $sq(c_i, R_j)$ is the product of the size of the source relation R_j and the selectivity of condition c_i in R_j. In the case of a semijoin query $sjq(c_i, R_j, X)$, the size of the result is the product of the size of R_j, the selectivity of c_i in R_j and the selectivity of the set of items X. The selectivity of a set of items X is approximated as the size of X divided by the total number of entities across all sources.

To illustrate the cost computations, consider a selection query to a source of size 10,000 with a condition of selectivity 0.05 for that source. This query will cost $1 + 1 + 5$ (a cost of 1 for the query, 1 for the result header, and 5 for the answer set of items, because there are $10,000 * 0.05 = 500$ answer items). Next, consider a semijoin query to the same source, with a semijoin set of 1,000 items. Let there be a total of 100,000 items across all sources. Of the $10,000 * 0.05 = 500$ items at the source that satisfy the condition, we expect $500 * (1,000/100,000) = 5$ items to join with the semijoin set sent as input. Thus, the cost of the semijoin query will be $1 + 1,000/100 + 1 + 5/100(\text{rounded up}) = 13$. If the source does not directly support semijoin queries, we need to send 1,000 selection queries, each costing $1 + 1 = 2$ units, for a total of 2,000 units.

Parameters. The main parameters of our model are the number and sizes of sources and the number and selectivities of the conditions. For the results we present here, we chose the following profile:

1. There are 10 sources and the query under consideration has 3 conditions. These are base values; the numbers are varied in the graphs that follow.

2. Half of the sources have 100 entries while the other half have 1,000,000 entries (thus, we model a mixture of source sizes).

3. One of the conditions has a selectivity of 0.01 while all others have a selectivity of 0.33 (thus, we model mixed selectivities). Each condition has the same selectivity at all the sources.

4.6.1 Comparing the Optimization Algorithms

The behavior of FILTER, SJ, and SJA is shown in Figure 4.7 and 4.8, as the number of sources and the number of conditions vary. It is clear from these graphs that the FILTER plans are dramatically more expensive than the other plans. In Figure 4.7, it appears that

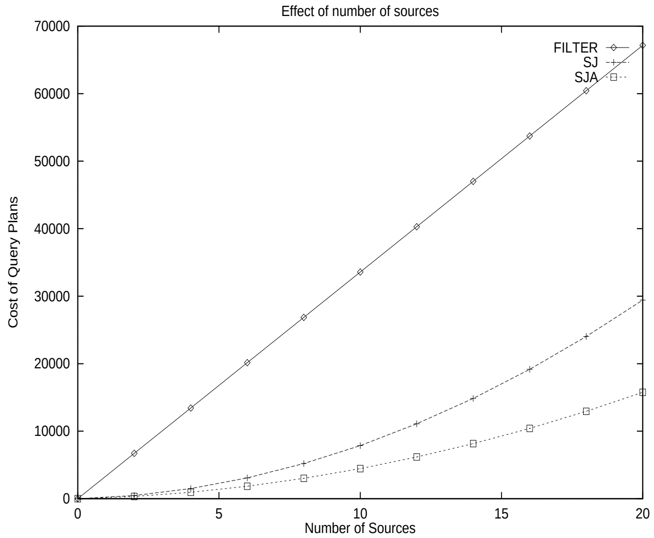

Figure 4.7: Varying Number of Sources.

the gap between FILTER and the others narrows as the number of sources increases, but even with 100 sources (not shown), FILTER does substantially worse.

The performance differences between the SJ and SJA plans is not as marked, but it is still significant. For instance, with more than 15 sources, or with 3 conditions or more, the SJA cost is roughly *half* of the SJ plan cost. Although not shown in these graphs, we observed that as source characteristics diverge (in terms of condition selectivities, cost functions), the performance advantages of SJA over SJ are magnified further. For instance, even if 10% of the sources do not support semijoin queries directly (and need expensive emulation at the mediator), the performance of SJ converges to that of FILTER, while the performance of SJA remains relatively unchanged.

We only present the results for varying numbers of sources and conditions. The results of experiments involving varying condition selectivities and source sizes (not shown here) showed similar trends.

Figure 4.9 illustrates the performance of SJA+ compared to the others, as the number of sources varies. In SJA+ plans, when a mediator loads an entire source (including non-merge attributes), we charge twice the cost per item read. In Figure 4.9, we see that the SJA+ postoptimization techniques yield significant performance improvements. For instance, with 10 sources, the cost of the SJA+ plan is about 40% of the cost of the SJA plan. More importantly, the cost of SJA+ plans increases linearly with the number of

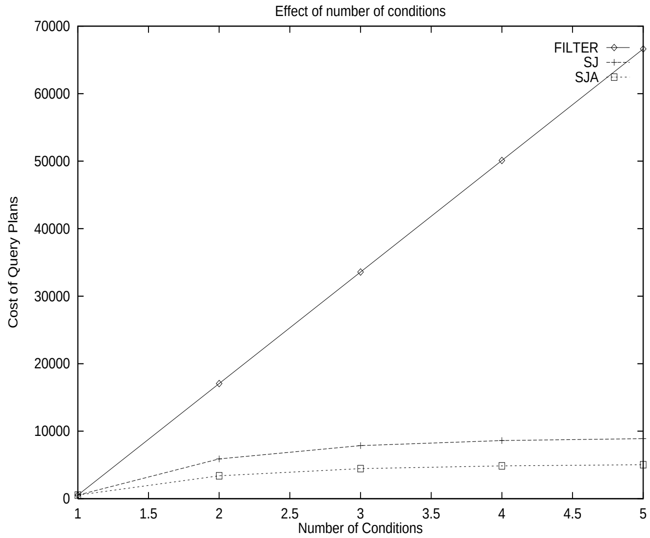

Figure 4.8: Varying Number of Conditions.

sources, so the advantage of SJA+ becomes significant for large numbers of sources. We also observed in our performance analysis that most of the advantages of SJA+ are due to its use of the difference operation, and not from the option of loading entire source contents.

In summary, our results (including many more not presented here) show that the performance differences between the various optimization algorithms can be significant over a wide range of parameter values. Thus, the extra cost of optimizing fusion queries carefully will very likely pay off.

4.6.2 Correlated Conditions

To illustrate the impact of condition correlations, we set up the following scenario. As before, we have 10 sources and 3 conditions; half of the sources have 100 entries and the other half have 1,000,000 entries. We set the selectivity of one condition to be 0.01 at all sources, while the selectivities of the other two conditions are set to be 0.5 at all sources. We factored in condition correlations by making the second and third condition dependent on the first. That is, we varied the conditional probability of an item satisfying the last two conditions, given that it satisfied the first condition. Varying this probability from 0 to 0.5 models negative correlations and from 0.5 to 1.0 models positive correlations. To magnify the effects of condition correlation we chose similar (both positive or both negative) correlations between the first and second conditions and between the first and

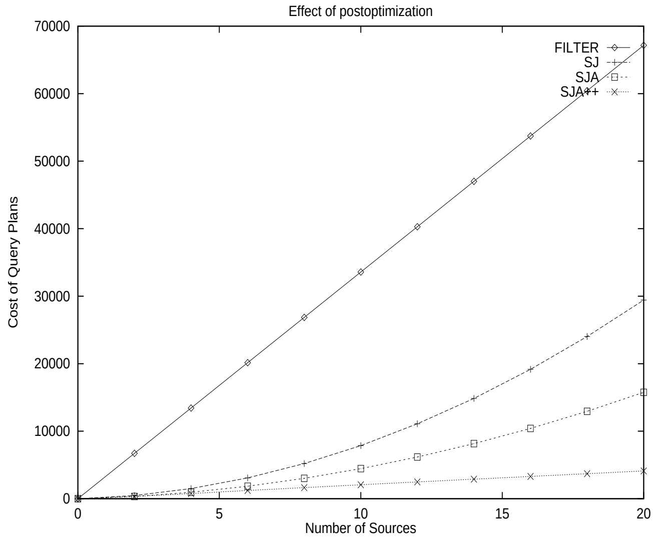

Figure 4.9: Postoptimization Improvements.

third conditions.

Figure 4.10 shows the impact of this conditional selectivity (varying between 0 and 1) on the costs of the plans generated by the FILTER, SJ, and SJA algorithms. (We do not consider SJA+ because of the difficulty in estimating its costs under this scenario.) Keep in mind that these optimization algorithms are not aware of the correlations; they generate the same plans as in the previous section.

The correlations do affect the actual costs, but as shown in Figure 4.10, the impact is minimal. The FILTER plans are completely unaffected because condition correlations do not affect selection queries to sources. Plans generated by SJ and SJA are affected, though, because semijoin queries are impacted by correlations. (Positive correlation will increase the size of the result of the semijoin query and may increase the cost, while negative correlation will decrease the size of the semijoin query and may decrease the cost.) However, the sensitivity to correlation is small: the cost of both SJ and SJA plans varies by about -5% to +5%. In fact, one has to look very carefully at the graph to see this 5% impact.

Other experiments we have conducted confirm that condition correlations rarely have a significant impact on the costs of fusion-query plans. Even when optimization algorithms use an invalid independence assumption, their performance, relative to other algorithms, and relative to what they could achieve knowing the correlations, does not change much. This observation implies that it may not be worth gathering condition-correlation information

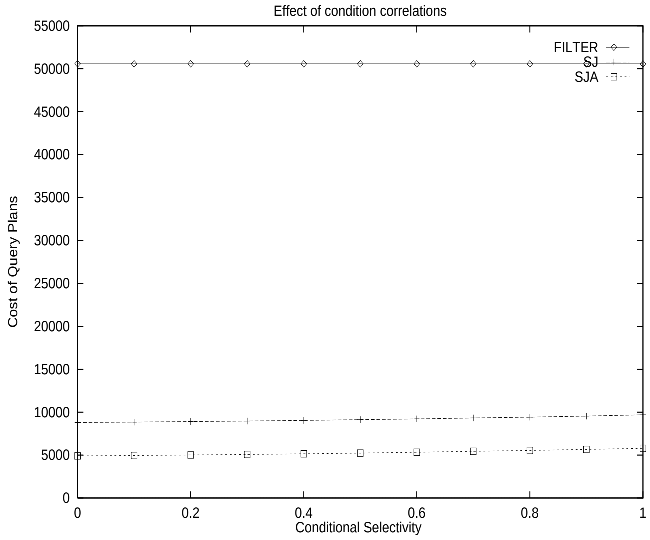

Figure 4.10: Correlated Conditions.

and incorporating it into the optimization process (through more accurate cost estimates), given that this task typically increases the cost of the optimization process by a significant amount.

Our final question has to do with whether the cost of the SJA plan shown in Figure 4.10 is far from that of the optimal simple plan. It turns out that in our scenario, the plan generated by SJA with no correlation information, is the same as the plan generated with cost estimates that know about the correlation, and is the same as the optimal simple plan. The latter is not surprising since the correlations in our scenario are intercondition, not intersource (see Section 4.4). So, at least our results confirm the intuition behind the conjecture at the end of section 4.4. In general, our results show that costs are rather insensitive to correlations, so we expect SJA (and SJA+) to generate excellent fusion-query plans, even in the face of condition correlations.

4.7 Fusion Queries in Existing Optimizers

The expansion of the Web has led to mediator prototypes that combine information from multiple heterogeneous sources [4, 9, 13, 18, 28, 37, 68, 71]. Similarly, prototypes for integrating databases have been developed, and recently integration products are being released or announced [2, 5, 7, 26, 45].

We note that there is a close connection between mediator-based systems [79] and distributed-database systems [53]. Many mediator systems have borrowed techniques for query optimization from the world of distributed databases. There has been a great deal of published work on these techniques [11, 53, 65]. However, most of this work focuses on the efficient evaluation of Select-Project-Join (SPJ) queries. It does not adequately address the special needs of fusion queries.

In this section we study how existing optimizers would handle fusion queries, and we explore opportunities for improvement based on the ideas developed in this chapter. The approaches taken by existing optimizers on fusion queries fall into three general categories, so we divide our discussion into three subsections. To illustrate, we continue to use the sample fusion query of Section 4.1, which finds all the drivers, from three DMV databases, that have "sp" and "dui" violations.

Distribution of the join over the union. The first category contains optimizers that distribute the join operation in a fusion query over the underlying unions. The corresponding query plan involves a union of SPJ subqueries, where each SPJ subquery can then be optimized using traditional methods. Such a plan for our sample fusion query is illustrated in Figure 4.11. In this plan, c_1 is *"$V = sp$"* and c_2 is *"$V = dui$"*. Each line describes a constituent SPJ subplan. The first line, for instance, is a plan for the SPJ subquery that finds the drivers that have both "sp" and "dui" infractions at R_1. The other lines (2 through 9) look for other possible ways to satisfy the query conditions, and the final three lines (10 through 12) union all the possible answers. Notice that there are an exponential number of ways to satisfy the query conditions, and correspondingly there are an exponential number of constituent SPJ subplans.

The plan of Figure 4.11 happens to be a filter plan. Optimizers in this category may also employ semijoin operations in their query plans. For example, the plan segment that computes Y_3 in the plan of Figure 4.11 may be replaced by $Y_3 := sjq(c_1, R_1, X_3)$. Consequently, we can have a simple plan that is not a filter plan. Note that plans like the one in Figure 4.11 are quite inefficient because they repeat many source queries. Common-subexpression elimination can be performed to generate more efficient plans. For example, we can eliminate the computation of $Y_4, Y_5, Y_6, Y_7, Y_8, Y_9, X_2, X_3, X_5, X_6, X_8$ and X_9 by reusing common subexpressions Y_1, Y_2, Y_3, X_1, X_4 and X_7. Thus, we end up with a more efficient filter plan, like the reference filter plans discussed in Section 4.2. When semijoin operations are used by

$$
\begin{array}{ll}
1) & X_1 := sq(c_1, R_1); Y_1 := sq(c_2, R_1); Z_1 := X_1 \cap Y_1 \\
2) & X_2 := sq(c_1, R_1); Y_2 := sq(c_2, R_2); Z_2 := X_2 \cap Y_2 \\
3) & X_3 := sq(c_1, R_1); Y_3 := sq(c_2, R_3); Z_3 := X_3 \cap Y_3 \\
4) & X_4 := sq(c_1, R_2); Y_4 := sq(c_2, R_1); Z_4 := X_4 \cap Y_4 \\
5) & X_5 := sq(c_1, R_2); Y_5 := sq(c_2, R_2); Z_5 := X_5 \cap Y_5 \\
6) & X_6 := sq(c_1, R_2); Y_6 := sq(c_2, R_3); Z_6 := X_6 \cap Y_6 \\
7) & X_7 := sq(c_1, R_3); Y_7 := sq(c_2, R_1); Z_7 := X_7 \cap Y_7 \\
8) & X_8 := sq(c_1, R_3); Y_8 := sq(c_2, R_2); Z_8 := X_8 \cap Y_8 \\
9) & X_9 := sq(c_1, R_3); Y_9 := sq(c_2, R_3); Z_9 := X_9 \cap Y_9 \\
10) & Z := Z_1 \cup Z_2; Z := Z \cup Z_3; Z := Z \cup Z_4 \\
11) & Z := Z \cup Z_5; Z := Z \cup Z_6; Z := Z \cup Z_7 \\
12) & Z := Z \cup Z_8; Z := Z \cup Z_9
\end{array}
$$

Figure 4.11: Distributing Join over Union.

the optimizers in this category, common-subexpression elimination becomes very expensive. This task takes time that is exponential in the number of the constituent SPJ subqueries, which in turn is exponential in the size of the original fusion query.

Examples of systems in our first category are Information Manifold [41], TSIMMIS [55], HERMES [68] and Infomaster [23]. Query processing in these systems is based on resolution [11, 24], which leads to the distribution of the join over the union. For example, Information Manifold would process our sample fusion query by generating nine datalog [75] subqueries, one for each of the nine constituent SPJ subqueries in Figure 4.11. It then unions the results of the nine datalog subqueries to form the answer to the given fusion query.

Systems taking this approach to fusion-query processing can incorporate techniques discussed in this chapter to implement a module that is specifically designed to handle fusion queries. This module can check if a query is a fusion query (by looking for the distinctive pattern of fusion queries) and use the algorithm (of Section 4.3) to generate a semijoin-adaptive plan for the identified fusion query. Thus, we can have a very efficient evaluation of fusion queries without incurring an extremely cumbersome and inefficient optimization process involving common-subexpression elimination.

Handling unions uniformly. The second approach to fusion-query processing is to separately process the union views, and conceptually generate "temporary" relations. Selection

$X_1 := sq(c_1, R_1)$	$X_1 := sq(c_1, R_1)$
$X_2 := sq(c_1, R_2)$	$X_2 := sq(c_1, R_2)$
$X_3 := sq(c_1, R_3)$	$X_3 := sq(c_1, R_3)$
$X := X_1 \cup X_2$	$X := X_1 \cup X_2$
$X := X \cup X_3$	$X := X \cup X_3$
$Y_1 := sq(c_2, R_1)$	$Y_1 := sjq(c_2, R_1, X)$
$Y_2 := sq(c_2, R_2)$	$Y_2 := sjq(c_2, R_2, X)$
$Y_3 := sq(c_2, R_3)$	$Y_3 := sjq(c_2, R_3, X)$
$Y := Y_1 \cup Y_2$	$Y := Y_1 \cup Y_2$
$Y := Y \cup Y_3$	$Y := Y \cup Y_3$
$Z := X \cap Y$	$Z := Y$
(a) A Filter Plan.	(b) A Semijoin Plan.

Figure 4.12: Handling Unions Uniformly.

conditions are applied as the temporary relations are computed. Then the temporary relations are joined. Examples of systems using such an approach are DB2 [22] and Tandem's NonStop SQL/MP [63].

Figure 4.12(a) shows a plan for our sample query generated by this class of optimizers. Here X and Y represent the "temporary relations." The query plans considered by these systems are characterized by the class of filter plans described in Section 4.2.

This approach of processing the unions uniformly, as described above, does not use semijoin operations in the query plans. A slight variation is to combine the steps of union view processing and the join processing. This variation includes semijoin operations. An example system that uses this variation is Tandem's NonStop SQL/MX [10].

A query plan for our sample query, using this variation, is shown in Figure 4.12(b). With this variation, the set of query plans considered includes the class of filter plans and the class of semijoin plans, but not the class of semijoin-adaptive plans. The various sources of a union view are treated homogeneously. That is, if two sources take part in a union view, they both get the same kind of source queries. The technique described in this chapter to find the best semijoin-adaptive plan for fusion queries can be employed by systems like Tandem's NonStop SQL/MX without a great deal of effort. The trick is to allow for heterogeneous treatment of the different elements of a union view. That is, one source in the union view may get a selection query while another source in the same union view may get a semijoin query.

Extensible optimizers Recent extensible optimizers [25, 27] use flexible, rule-based approaches. The key to efficient fusion-query processing in these systems lies in the set of rules defined. We believe that one can write rules, which embody our techniques, to carry out fusion-query processing in these optimizers. For example, it is easy to write rules in the Garlic system [28] that help generate efficient filter plans for fusion queries. Combining these with rules for semijoin operations, like the ones given in [28], we can generate semijoin plans for fusion queries. We believe that an extension of these rules (nontrivial, but perhaps not very difficult) may yield semijoin-adaptive plans for fusion queries. Another way to incorporate our techniques into optimizers following the rule-based approach is to have a rule that identifies a fusion query and generates the best semijoin-adaptive plan for it.

4.8 Chapter Summary

Fusion queries are important in environments where data is not well organized and partitioned across autonomous, distributed sites. In this chapter we have developed a formal framework for optimizing fusion queries, and we have provided efficient algorithms to produce good query plans over broad scenarios. We have also described enhancements (postoptimizations) that can boost performance significantly, with relatively little additional optimization cost. As discussed in Section 4.7, our results can be useful for understanding the types of plans current optimizers generate for fusion queries, as well as for improving their performance.

Finally, note that we focused on minimizing the total work in executing a query. One could also consider minimizing the *response time* of a query in a parallel execution model. It is clear that FILTER plans can easily take advantage of parallelism. However, the plans selected by SJ and SJA can also benefit significantly from parallel execution. SJ and SJA generate query plans that process one condition at a time. For each condition, they issue queries to all the sources. If the source queries are issued in parallel, the time to process a particular condition will be the time taken by the longest source query for that condition.

There are also opportunities for pipelining source queries. For FILTER plans, the pipelining of the queries on each source thread is simple. Pipelining may be more complicated in the case of SJ and SJA, but still possible. For instance, SJA pipelining can be achieved as follows: a mediator process in charge of condition c_i receives streams of items satisfying $c_1, ..., c_{i-1}$ from the various sources; whenever it has obtained a batch of new items, it sends them to all sources that are expecting a semijoin set to handle c_i. Some simple experiments we conducted indicate that SJA, issuing queries in parallel but without pipelining, maintains a large response-time advantage over FILTER, with pipelining on parallel source threads. More experiments and a more comprehensive study of optimization of fusion queries in parallel execution models are needed for a better understanding of all the important issues.

Chapter 5

Computing Mediator Capabilities

5.1 Overview

Contemporary mediator systems such as TSIMMIS [13, 55], Information Manifold [37, 41], Garlic [9, 28], and DISCO [35, 71] perform capability-based query processing. Sources express their capabilities in these systems through a variety of mechanisms – query templates, capability records, and simple capability-description grammars. However, none of these systems computes the query-capabilities of mediators based on the supported source queries. Not having the mediator capabilities readily available makes it difficult to treat mediators as sources for other mediators. Furthermore, users have a difficult time understanding the set of supported mediator queries in these systems. Consequently, users must endure a frustrating trial-and-error approach, submitting queries that are rejected until finally hitting upon a query that is answered by the mediator.

In this chapter, we present algorithms to precompute mediator capabilities automatically, so that users and other mediators know which queries are supported.[1] Computing mediator capabilities can be simple in some cases. For instance, consider a mediator like the Yahoo shopping guide for books [97], which offers a union view on top of a set of views provided by Web bookstores. We may be able to combine all the restrictions of the sources in an easy manner to arrive at the capabilities of the mediator. Let one bookstore allow search by specifying any one of author, title and subject attributes, while another bookstore allows search by specifying any combination of author and title attributes. The combined

[1]The material presented in this chapter is based on ideas developed by the author with Chen Li, Hector Garcia-Molina and Jeff Ullman. Some of these ideas had been presented in the conference paper [82].

set of restrictions of these two bookstores suggests that the Yahoo shopping guide can support queries that search for books by specifying the author or the title attribute. Queries specifying the subject attribute alone are not feasible because the second bookstore does not answer this query.

In many cases, the computation of mediator capabilities is more complicated due to the rich variety of source-capability limitations and the host of techniques mediators employ in processing queries. For instance, as we have seen in Chapter 2, by using special techniques to postprocess the results of queries on weaker data sources, mediators can support much larger sets of queries. Thus, the capabilities we compute for mediators can be "stronger" than those of the underlying sources.

Another complication arises when we consider mediator capabilities that depend on the contents of sources. In such cases, a mediator may define its capabilities with a "lower bound" that specifies which queries will always be answerable, and an "upper bound" indicating which queries will never be answerable. Queries "between" bounds may be answerable, depending on the contents of the sources at run time. We refer to mediators that handle upper and lower bounds as *dynamic mediators*. In Section 5.5, we discuss how dynamic mediators process queries and how their capabilities can be determined.

Deriving mediator forms by hand is error-prone. Manual computation of mediator capabilities is also expensive, since each time a source changes its capabilities, we may have to update the mediator capabilities. We need to develop algorithms to compute mediator capabilities automatically.

In this chapter, we develop the following:

- A framework for describing the capability limitations of data sources. Our framework allows us to compute and describe the corresponding capabilities of mediators built on such data sources. In our framework, we model restrictions on attribute specification in queries as well as restrictions on attribute output in the results of queries.
- Algorithms for computing the query capabilities of mediators based on the capabilities of the sources they mediate.
- Advanced techniques that could be used by dynamic mediators to process queries beyond those that are feasible in conventional mediators.

We start with a mediator specification that defines integrated views in terms of source views. We assume that differences in ontologies, vocabularies, and formats used by sources have been resolved. In particular, if two sources share an attribute name, we assume that the attributes are equivalent (i.e., wrappers take care of any differences). Related research [46, 56] suggests ways to deal with ontology and format differences. The problem we are addressing in this chapter specifically deals with the computation of mediator capabilities, after the mediator views and their definition in terms of source views are constructed.

5.2 Framework

In this section, we present our framework for describing the query capabilities of data sources and mediators. In our framework, each data source exports a set of relational views.[2] Conceptually, a query to a source is submitted by filling out a form on one of the views exported by the source. The query specifies values for some attributes of the view, and the result of executing the query is a set of tuples of the source view on which the query is posed. The following example illustrates a source view and answering queries on that view.

EXAMPLE 5.2.1 Consider a source that exports a view $R(X, Y, Z)$. Let the set of tuples in source view R be $\{(x_1, y_1, z_1), (x_1, y_2, z_1), (x_2, y_2, z_2)\}$. The result of the source query $R(X, Y, z_1)$ is $\{(x_1, y_1, z_1), (x_1, y_2, z_1)\}$, while the result of the source query $R(X, y_1, Z)$ is $\{(x_1, y_1, z_1)\}$. Other sample queries include $R(x_1, Y, z_1)$, $R(x_1, Y, Z)$ and $R(X, Y, Z)$. □

5.2.1 Mediator Views

A mediator integrates data from multiple sources and exports a set of integrated views. Each integrated view is defined in terms of source views and/or other integrated views of the mediator. The set of operations used to define integrated views are *union*, *join*, *selection* and *projection*. We do not consider recursive views, so the mediator-view definitions form a directed acyclic graph (DAG). Since we allow the use of integrated views to define other integrated views, we can assume without loss of generality that each definition of a mediator view uses only one operator – union, join, selection or projection.

[2] We use the relational framework for simplicity of exposition. We believe that all the main ideas presented in this chapter carry over seamlessly to other data models like OEM [57] and XML [96]. In fact, our interest in computing mediator capabilities is based on our work in the TSIMMIS project, which uses the OEM data model.

EXAMPLE 5.2.2 Consider a mediator with five data sources. Let the source views be $R_1(X,Y,Z)$, $R_2(X,Y,Z)$, $R_3(X,Y,Z)$, $R_4(Z,U)$, and $R_5(U,V,W)$. Let the mediator define the following two views: $M_1(X,Y,Z)$ is the union of R_1, R_2 and R_3, $M_2(X,Y,Z,U,V,W)$ is the join of$M_1(X,Y,Z)$, R_4 and R_5. Users can pose queries on the views of a mediator in a manner similar to submitting queries on source views. For instance, one can specify $M_2(x_1,Y,Z,U,V,w_1)$ as a query to the mediator. □

5.2.2 Query Templates

The query capabilities of a data source are expressed as a set of *templates* supported by the source. A template, like a form on the Web, identifies the various attributes of a source view that can be specified in a query submitted on the view. Restrictions on attribute specification are also indicated by templates (e.g., if an attribute value has to be chosen from a menu of choices). We use attribute *adornments* to specify how the attributes of a view can participate in a query template.

Based on our study of query forms for a variety of Web data sources, we consider the following five kinds of attribute adornments:

1. adornment **f** (for free): the attribute may or may not be specified in the query;
2. adornment **u** (for unspecifiable): the attribute cannot be specified in the query;
3. adornment **b** (for bound): the attribute must be specified in the query;
4. adornment $\mathbf{c}[S]$ (for constant): the attribute must be specified, and in addition, it must be chosen from the set of constants S;
5. adornment $\mathbf{o}[S]$ (for optional): the attribute may or may not be specified in the query, and if specified, it must be chosen from the set of constants S.

Each template specifies an adornment for each attribute of a view. In the case of the c and o adornments, the menus of constants allowed are also specified by the templates. The following example illustrates how templates with attribute adornments are used to express query capabilities.

EXAMPLE 5.2.3 Consider the source of Example 5.2.1 that exports the single view $R(X,Y,Z)$. Let the source support a set of queries on this view that specify some value for

X and no value for Y. In addition, let these queries optionally specify some value for Z. A template that expresses these query capabilities is *buf*. This template is similar to a Web form in which only the X and the Z fields appear, with the requirement that the X field must be specified when submitting the form.

Suppose the source also supports another set of queries in which X cannot be specified at all, Y can be specified optionally while Z must be specified and must be chosen from $\{z_1, z_2\}$. These query capabilities can be expressed by the template *ufc[*z_1, z_2*]*.

Based on the two templates, one can easily determine whether a given query on the source view is answerable or not. For instance, $R(x_1, Y, Z)$ is an answerable query because it satisfies the first template, while $R(X, Y, z_1)$ is an answerable query because it satisfies the second template. $R(X, y_1, Z)$ and $R(X, Y, z_3)$ do not satisfy either template. So, they are not answerable queries. □

We use the same mechanism of adorned templates to describe mediator capabilities. That is, the capabilities of a mediator are expressed as a set of templates on the views exported by the mediator. In this chapter, we study the process of computing the templates of mediator views based on the templates of source views.

As noted earlier, templates indicate restrictions on attribute specification in queries to sources and mediators. In Section 5.6, we show how templates can also be used to describe restrictions on the set of attributes returned when mediators and sources answer queries. We discuss there the effect of output restrictions on computing the capabilities of mediators. Until then, we do not consider output restrictions in the computation of mediator templates, as presented in Sections 5.3 and 5.4.

5.3 Templates for Simple Mediators

The set of answerable queries of a mediator is affected by the techniques used by the mediator in processing the queries posed to it. We start by considering the following simple query-processing scheme: When a query is submitted, the mediator translates this query into a set of relevant source queries by transferring the query bindings; subsequently, the mediator combines the results of the source queries, based on the operations (union, join, selection, projection) appearing in the definition of the mediator view on which the query is submitted. In particular, we make the following assumptions:

- Mediators do not apply any postprocessing conditions, other than those listed explicitly in the view definitions.
- Mediators perform join operations locally, i.e., they do not pass bindings from one join operand to another.

The simple query-processing scheme we described above is likely to be supported by most mediators, so we choose it for our base case. We call mediators employing this query-processing scheme *simple mediators*. In Section 5.4, we discuss how mediators can employ additional postprocessing techniques and join methods to enhance their sets of answerable queries.

5.3.1 Union Views

We start by computing the set of templates for a union view. We present this computation in three steps. The first step deals with the simple case of a union view that has two base views, with each base view having exactly one template. Next, we show how to compute the set of templates with two base views, but with each base view having an arbitrary number of templates. Then, we describe the computation for a union view with an arbitrary number of base views, each having an arbitrary number of templates.

Union of Two Single-Template Base Views. For each attribute, we compute its adornment in the union-view template based on its adornments in the two base-view templates. This computation is based on the mapping function presented in Table 5.1. Given the two adornments of an attribute in the two base-view templates, the table indicates the adornment of the attribute in the union-view template. For example, the entry in the table for the combination of f and b is b because the b adornment in one of the base-view templates forces the mediator to require that the attribute must be specified in the union-view query. Note that the mapping function of Table 5.1 is symmetric.

When the adornments of an attribute in the base-view templates involve menus of constants, we need to compute the resulting menu of constants. As indicated in Table 5.1, when only one of the base-view templates has a menu (with adornment o or c), this menu is copied over to the union-view template. When both the base-view templates have menus, we intersect the two menus.

In some cases, the base-view adornments of an attribute cannot be combined to arrive at a valid union-view adornment. Such cases are indicated by "-" in Table 5.1. For instance,

	f	**o**$[S_{21}]$	**b**	**c**$[S_{22}]$	**u**
f	f	o$[S_{21}]$	b	c$[S_{22}]$	u
o$[S_{11}]$	o$[S_{11}]$	o$[S_{11} \cap S_{21}]$	c$[S_{11}]$	c$[S_{11} \cap S_{22}]$	u
b	b	c$[S_{21}]$	b	c$[S_{22}]$	-
c$[S_{12}]$	c$[S_{12}]$	c$[S_{12} \cap S_{21}]$	c$[S_{12}]$	c$[S_{12} \cap S_{22}]$	-
u	u	u	-	-	u

Table 5.1: Mapping Function for Combining Adornments from Two Templates.

consider the case of one base view having the b adornment and the other having the u adornment. No valid adornment can be computed in this case because the b adornment of the first base view forces the mediator to require that the attribute must be specified, while the u adornment of the other base view prevents the mediator from allowing the attribute to be specified in the union-view query.[3] Although not shown explicitly in Table 5.1, an attribute may also end up with an invalid adornment when computing its menu of constants by intersecting its menus from the base-view templates. In particular, the $c[S]$ entries in Table 5.1 are replaced by "-" when S is empty. Note that the $o[S]$ entry should be replaced by u when S is empty.

During the computation of the union-view template, if any attribute is determined to have the invalid adornment, we declare that no union-view template can be computed from the two base-view templates.

Union of Two Base Views with Multiple Templates. For each pair in the cross product of the template sets of the two base views, we compute a template for the union view based on the process described above. As noted earlier, in some cases no union-view template can result from a pair of base-view templates. Accordingly, the number of templates for a union view over two base views with template sets T_1 and T_2 varies from zero to $|T_1| \times |T_2|$.

Union of Multiple Base Views with Multiple Templates. We compute the templates of the union view by considering two base views at a time. That is, if the union view has n base views, we invoke the method of computing templates for a union view with two base views $(n-1)$ times. Note that the associativity and commutativity of the mapping function presented in Table 5.1 allow us to carry out the computation in this simple manner.

[3] Recall that we are dealing with simple mediators in this section. In the next section, we will see how mediators use postprocessing techniques to arrive at a valid adornment of b when the base view adornments are b and u.

EXAMPLE 5.3.1 Let a mediator view $M(X, Y, Z)$ be defined as the union of three source views $R_1(X, Y, Z)$, $R_2(X, Y, Z)$ and $R_3(X, Y, Z)$. Let R_1 have two templates: *bff* and *ffb*, let R_2 have the single template *fbf*, and let R_3 have two templates: *ffc[*S_1*]* and *c[*S_2*]ff*.

When computing the templates of $R_1 \cup R_2$, we consider two combinations of base-view templates: *bff* for R_1 and *fbf* for R_2; *ffb* for R_1 and *fbf* for R_2. Based on the first combination, we compute the template *bbf*, and the second combination yields the template *fbb*.

Now, for $(R_1 \cup R_2) \cup R_3$, four combinations of templates are considered and they result in the following four templates for M: *bbc[*S_1*]*, *fbc[*S_1*]*, *c[*S_2*]bf* and *c[*S_2*]bb*. □

5.3.2 Join Views

As noted earlier, a simple mediator processes a query on a join view by first transferring the query bindings to the base views, and then joining the results of the base-view queries. There is one major difference between computing templates for join views and computing templates for union views: Some of the attributes in a join view may not appear in all its base views. Accordingly, the adornment computation of the join attributes is different from that of the nonjoin attributes.

Join of Two Single-Template Base Views. For all nonjoin attributes, we copy over their adornments from the base-view templates (each nonjoin attribute appears in exactly one of the base views). For each join attribute, the adornment computation employs the same mapping function (see Table 5.1) used by the union-view template computation.

Join of Two Base Views with Multiple Templates. As in the union case, for each pair in the cross product of the sets of base-view templates, we compute a join-view template.

Join of Multiple Base Views with Multiple Templates. Once again, as in the union case, we consider two base views at a time. If the join view has n base views, we invoke the method of computing templates for a join view with two base views $(n-1)$ times.

EXAMPLE 5.3.2 Let a mediator view $M(X, Y, Z, U, V, W)$ be defined as the join of three base views $R_1(X, Y, Z)$, $R_2(Z, U, V)$ and $R_3(V, W)$. Let R_1 have two templates: *fbf* and *bff*, let R_2 have two templates: *bfb* and *fc[*S_1*]f*, and let R_3 have the single template *fb*. The join of R_1 and R_2 results in four templates: *fbbfb*, *fbfc[*S_1*]f*, *bfbfb* and *bffc[*S_1*]f*. When we join in R_3, the four templates are modified as: *fbbfbb*, *fbfc[*S_1*]fb*, *bfbfbb* and *bffc[*S_1*]fb*. □

5.3.3 Selection Views

When processing a query over a selection view, the mediator copies it over into a query on the underlying base view and applies the selection predicate on the results of the base-view query. Therefore, the set of base-view templates are simply copied over as the set of selection-view templates.

EXAMPLE 5.3.3 Let $M(X, Y, Z)$ be a selection view with $R(X, Y, Z)$ as its base view and $(X < x_1)$ as the selection predicate. Let R have the template *fbu*. Then, M also has the template *fbu*. For instance, one can pose a query on M by specifying a value for the attribute Y, like $M(X, y_1, Z)$. The mediator can copy over this query on M into the query $R(X, y_1, Z)$, and apply the predicate $(X < x_1)$ on the result of the query on R. □

Note that we can be "smarter" than simply repeating the adornments of the base-view templates in the selection-view templates. For instance, if a source does not allow queries specifying a particular attribute (i.e., it has the *u* adornment for this attribute), the mediator could allow more flexible queries by processing extra predicates on that attribute, along with processing the selection-view predicate on the results of the corresponding source queries. Such special postprocessing of source-query results represents more mediator processing than is considered by the base case of our simple mediators. This topic is further discussed in Section 5.4.

5.3.4 Projection Views

A query on a projection view is translated into a query on the underlying base view by simply leaving the hidden attributes (those that are in the base view but not in the projection view) unspecified. If any of the hidden attributes has a *b* or *c* adornment in a base-view template, the translated query does not match this base-view template. Therefore, we only create a template for the projection view whenever the base-view template has the *f*, *o* or *u* adornments for the hidden attributes. The created projection-view template simply copies over the adornments for each of the projected attributes from the base-view template.

EXAMPLE 5.3.4 Consider a base view $R(X, Y, Z)$ and a projection view $M(X, Z)$ defined over it. Let R have the following templates: *fbu*, *bfu*. The first template of R does not yield any template of M because the hidden attribute Y has the *b* adornment. From the second template of R, we compute the *bu* template of M. That is, M can support queries that bind the X attribute and that leave the Z attribute unspecified. □

5.3.5 Templates for Graph of Views

So far, we have presented the machinery to compute the templates of a single mediator view starting from the templates of its base views. We can extend this machinery in a straightforward manner to compute the templates of all the views at a mediator. In particular, we obtain the templates of the views exported by the mediator and thus ascertain the set of queries supported by the mediator.

In the view graph of the mediator there is an edge from V_1 to V_2 whenever V_2 is used in defining V_1. Recall that the view graph at a mediator is a directed acyclic graph with the source views having no outgoing edges. Based on the techniques described above, we compute the templates of all the views in the graph respecting the order constraints imposed by the edges of the graph. That is, we choose a topological sorting of the views and compute their adornments in that order.

Whenever the mediator specification is modified, we can recompute the templates of only those views that are affected by the changes. A mediator view is affected by the changes if its definition is altered or if its definition contains a view that is affected by the changes.

5.4 Advanced Query-Processing Techniques

In this section, we consider some techniques used by mediators to support more queries than are supported by the simple mediators of the previous section. In Chapter 2, we discussed how mediators use postprocessing techniques to support query conditions that sources do not. We presented in Chapter 3 the technique of passing bindings that mediators use to process join queries. The following examples recall those techniques and illustrate their effects on the computation of templates for mediator views.

EXAMPLE 5.4.1 Let a mediator view $M(X, Y, Z)$ be the union of two source views $R_1(X, Y, Z)$ and $R_2(X, Y, Z)$. Suppose R_1 has the single template *bfu*, and R_2 has the single template *buf*. In the case of a simple mediator, we compute the single template *buu* for M. Based on this template, $M(x_1, Y, Z)$ is a feasible query, while $M(x_1, y_1, z_1)$ is not.

If the mediator can postprocess the results of queries on the underlying views, then it can support more queries. For instance, it can support the query $M(x_1, y_1, z_1)$ by first invoking the feasible queries $R_1(x_1, y_1, Z)$ and $R_2(x_1, Y, z_1)$, and then filtering the results of these queries with respect to the conditions on the Y and the Z attributes. In particular, it can apply the condition $(Z = z_1)$ on the result of $R_1(x_1, y_1, Z)$ and the condition $(Y = y_1)$ on the result of $R_2(x_1, Y, z_1)$. The union of the postprocessed results of source queries gives the answer to the query $M(x_1, y_1, z_1)$. Thus, the ability of the mediator to postprocess the results of underlying queries can enhance the set of feasible queries supported by the mediator. □

EXAMPLE 5.4.2 Let a mediator view $M(X, Y, Z, U, V)$ be the join of two source views $R_1(X, Y, Z)$ and $R_2(Z, U, V)$. Suppose R_1 has the single template *bfb*, and R_2 has the single template *fub*. In the case of a simple mediator, M has the single template *bfbub*. For instance, $M(x_1, Y, z_1, U, v_1)$ is a feasible query while $M(x_1, y_1, Z, U, v_1)$ is not.

If the mediator can perform join operations by passing bindings from one join operand to the next (i.e., it can perform *bind joins* instead of the simple *local joins* of Section 5.3), the query $M(x_1, y_1, Z, U, v_1)$ can be answered. The mediator can first execute the query $R_2(Z, U, v_1)$ and pass bindings for the Z attribute from the result of this query to the query on R_1. That is, for each value of Z, say z_i, in the result of $R_2(Z, U, v_1)$, the mediator can invoke the query $R_1(x_1, y_1, z_i)$. The answer to the query on M is obtained from the results of all these R_1 queries and the R_2 query. Thus, the ability to perform join operations by passing bindings enhances the set of queries a mediator can support. □

Based on the techniques discussed above, we will re-examine the computation of mediator templates to identify the larger sets of feasible queries. Specifically, we consider union, join, selection and projection views, and describe how their template computations differ from those of the previous section.

5.4.1 Union Views

As before, the computation of union-view templates can be described in three steps. In fact, the only difference between this computation and the one in Section 5.3.1 is in the first step, where we compute the template of a union view defined over two base views, each with a single template.

	f	**o**$[S_{21}]$	**b**	**c**$[S_{22}]$	**u**
f	f	f	b	c$[S_{22}]$	f
o$[S_{11}]$	f	f	b	c$[S_{22}]$	f
b	b	b	b	c$[S_{22}]$	b
c$[S_{12}]$	c$[S_{12}]$	c$[S_{12}]$	c$[S_{12}]$	c$[S_{12} \cap S_{22}]$	c$[S_{12}]$
u	f	f	b	c$[S_{22}]$	f

Table 5.2: Mapping Function for Union with Postprocessing.

Union of Two Single-Template Base Views. Using the same notation as before, we define a new mapping function for the computation of an attribute's adornment in the union-view template from the attribute's adornments in the two base-view templates. The new mapping function is shown in Table 5.2.

The essential difference between the mapping function of Table 5.2 and the mapping function used in Section 5.3.1 is in the treatment of the u and the o adornments. When a base-view adornment is u, the mediator can invoke a query on this view without specifying a value for this attribute and then optionally support a value specified by the union-view query for this attribute in the postprocessing step. Therefore, the u adornment is treated the same way as the *f* adornment. In a similar way, when a base-view template has the o adornment for an attribute, and a value specified by the union-view query for this attribute is not one of the menu constants associated with the o adornment, the mediator can execute a query on the base view without specifying any value for this attribute and check for the value indicated by the union-view query in the postprocessing step. Thus, o is also treated as *f*.

EXAMPLE 5.4.3 Let a mediator view $M(X, Y, Z)$ be defined as the union of two source views $R_1(X, Y, Z)$ and $R_2(X, Y, Z)$. Let R_1 have two templates: *bfo*$[S_1]$ and *o*$[S_2]$*fb*. Let R_2 have the single template *fbu*.

To compute the templates for M, we consider two combinations of base-view templates: *bfo*$[S_1]$ and *fbu*, *o*$[S_2]$*fb* and *fbu*. From the first combination, we arrive at the template *bbf* for M. The second combination results in the template *fbb*.

Note that if the mediator does not perform postprocessing, we will only have one template for M: *bbu*. Besides, this template is more restrictive than the *bbf* template that we obtained above by using mediator postprocessing. □

	f	**o**$[S_{21}]$	**b**	**c**$[S_{22}]$	**u**
f	f	f	f	c$[S_{22}]$	f
o$[S_{11}]$	f	f	f	c$[S_{22}]$	f
b	b	b	b	c$[S_{22}]$	b
c$[S_{12}]$	c$[S_{12}]$	c$[S_{12}]$	c$[S_{12}]$	c$[S_{12} \cap S_{22}]$	c$[S_{12}]$
u	f	f	f	c$[S_{22}]$	f

Table 5.3: Mapping Function for Join by Passing Bindings.

5.4.2 Join Views

When processing a query on a join view over a set of base views, since the attribute values returned from one base-view query can be used to satisfy the binding requirements of the subsequent base-view queries, the order in which the base-view queries are considered is important. Accordingly, the template computation for join views is presented in four steps. The first step considers a join sequence of two base views, each with one template. Then we deal with a join sequence of two base views, each with an arbitrary number of templates. Next we handle a join sequence of an arbitrary number of base views, each with an arbitrary number of templates. Finally, in the fourth step, we compute the templates of a join view by considering all the sequences of its base views.

Join Sequence of Two Single-Template Base Views. As before, for all nonjoin attributes we copy over their adornments from the base-view templates. The mapping function for computing the adornment of a join attribute is presented in Table 5.3. The adornment of the first base-view is listed on the left and the adornment of the second base-view is listed on the top of the table. Because the mediator can perform joins by passing bindings, the case of a b adornment for the second base view is similar to the case of an f adornment. The mediator can pass the required binding for the second base view from the result of the query on the first base view.

Join Sequence of Two Base Views with Multiple Templates. As in the union case, we repeatedly invoke the method that handles single-template base views and compute the set of templates of the join sequence. That is, for each combination of base-view templates, we obtain a template for the join sequence.

Join Sequence of Multiple Base Views with Multiple Templates. We associate left-to-right the base views in the join sequence (note that Table 5.3 is associative, but not symmetric). That is, if the join sequence has n base views, we call the method of computing templates for a join sequence of two base views $(n-1)$ times.

Join View of Multiple Base Views with Multiple Templates. We consider all the possible sequences of the base views, and for each sequence, we invoke the above method to compute a set of templates. For a join view with n base views, we call this method $n!$ times. We take the union of the $n!$ resulting sets of templates to arrive at the set of templates for the join view.

EXAMPLE 5.4.4 Let a mediator view $M(X, Y, Z, U, V, W)$ be defined as the join of three base views $R_1(X, Y, Z)$, $R_2(Z, U, V)$ and $R_3(V, W)$. Let R_1 have two templates: *fbf* and *bfb*, let R_2 have two templates: *bfb* and *fbf*, and let R_3 have the single template *fb*.

We consider a total of six sequences. For the sequence $\langle R_1, R_2, R_3 \rangle$, we end up with four templates: *fbffbb*, *fbfbfb*, *bfbfbb* and *bfbbfb*. For the sequence $\langle R_3, R_2, R_1 \rangle$, the templates are: *fbbffb*, *fbfbfb*, *bfbffb* and *bffbfb*. Continuing in this manner, we can compute another four sets of four templates each, based on the remaining four sequences. The union of these six sets of templates yields a set of twenty-four templates for M. After eliminating duplicates in this set, we end up with the following set of nine templates: *fbffbb*, *fbfbfb*, *bfbfbb*, *bfbbfb*, *fbbffb*, *bfbffb*, *bffbfb*, *fbfffb* and *fbbfbb*.

Without the ability to perform joins by passing bindings, the set of templates for the mediator view M would be limited to: *fbbfbb*, *fbfbfb*, *bfbfbb* and *bfbbfb*. Notice that these four templates are more restrictive than the nine templates we obtained for a mediator that passes bindings. In particular, the set of queries covered by the nine templates of M supported by the mediator through bind joins is a strict superset of the set of queries covered by the four templates obtained through local joins. For instance, the query $M(X, y_1, Z, U, v_1, w_1)$ is feasible only if the mediator passes bindings. □

5.4.3 Selection and Projection Views

For selection views, the template computation is quite different from that of Section 5.3.3. As before, we do generate a selection-view template corresponding to each of its base-view templates. However, we do not simply copy over the adornments of attributes from the base-view templates to the selection-view templates.

EXAMPLE 5.4.5 Let $M(X, Y, Z)$ be a selection view with $R(X, Y, Z)$ as its base view and $(X < x_1)$ as the selection condition. Let R have a template *bfu*. According to the template computation in Section 5.3.3, the mediator has the template *bfu* for M.

The *bfu* template of M does not allow a query like $M(x_2, Y, z_1)$. However, the mediator can make use of its postprocessing abilities to support such a query. Given $M(x_2, Y, z_1)$, the mediator can first process the feasible base-view query $R(x_2, y_1, Z)$ and then filter the results of this query with the condition $(Z = z_1)$. Therefore, M can have the more flexible template *bff*.

Now, let us suppose that the selection predicate on M is $(X = x_1)$ instead of $(X < x_1)$. The *bff* template of M precludes a query like $M(X, y_1, Z)$. However, the mediator can infer from the selection-view predicate that it can translate $M(X, y_1, Z)$ into $R(x_1, y_1, Z)$, a feasible query on the base-view. Thus, it can support the query $M(X, y_1, Z)$. To reflect this ability to support "additional" queries on M, the template of M is changed to *fff*. In this way, a mediator that performs postprocessing can have selection-view templates that are much more flexible than their corresponding base-view templates. □

Base-View Adornment	Selection-Attribute Adornment	Nonselection-Attribute Adornment
f	f	f
o$[S_1]$	f	f
b	f or b	b
c$[S_1]$	f or c$[S_1]$	c$[S_1]$
u	f	f

Table 5.4: Mapping Function for Selection with Postprocessing.

Base-View Adornment	Projection-View Adornment
f	f
o$[S_1]$	f
b	b
c$[S_1]$	c$[S_1]$
u	f

Table 5.5: Mapping Function for Projection with Postprocessing.

The new rules for the computation of the selection-view templates are based on the mapping function given in Table 5.4. We consider two cases for each attribute in the selection view: (i) the selection predicate specifies a value for the attribute; (ii) the selection predicate does not specify a value for the attribute. In both cases, by employing postprocessing operations at the mediator, base-view adornments of o and u are converted to the selection-view adornment of *f*. In addition, the *b* adornment is also converted to the less restrictive *f* adornment if a value for the attribute can be inferred from the selection predicate. A *c* adornment is converted into an *f* adornment if the selection predicate specifies a value for the attribute that is in the constant set associated with the attribute's adornment in the base-view template. If the inferred value is not in the set of constants of the base-view template, we simply copy over the *c* adornment to the selection-view template.

The computation of templates for projection views is similar to that of Section 5.3.4, except that *u* and *o* adornments in the base-view templates are treated as *f* adornments. As shown in Table 5.5, we copy over the *f*, *b* and *c* adornments of the projected attributes from the base-view templates to the corresponding projection-view templates, while the *u* and *o* adornments for the projected attributes are changed to the *f* adornment in the projection-view templates. As before, we do not derive a projection-view template from a base-view template that has the *b* or the *c* adornment for a hidden attribute.

The above discussion about computing templates for projection views leads to a simplified way to understand the adornment mapping tables of union views (presented earlier in this section). One could first consider a simple projection of the base views onto the entire schemas (i.e., all attributes are projected) and then apply the method of Section 5.3 for computing the union-view templates.

5.5 Dynamic Mediators

So far, we have seen how the templates of a mediator can be computed in order to specify the set of queries answerable by the mediator. Given that computation, a query is supported by the mediator if it satisfies one of the mediator templates. However, as illustrated by Example 5.5.1, it may not be necessary for a query to satisfy a template in order for it to be answerable.

EXAMPLE 5.5.1 Consider a mediator view $M(X, Y, Z)$ defined as a join of two source views $R_1(X, Y)$ and $R_2(Y, Z)$. Let R_1 have the single template *bf* and let R_2 have the single template *c[S]f*, where S is $\{y_1, y_2, y_3\}$. Based on the computation described in Section 5.4, M has the single template *bc[S]f*. Query $M(x_1, y_1, Z)$ is answerable because it satisfies the template of M.

Consider the query $M(x_1, Y, Z)$. This query does not satisfy the template of M. However, the mediator may attempt to process the query anyway. It can perform a bind join by processing the query $R_1(x_1, Y)$ and passing bindings for the Y attribute in the queries to R_2. The set of Y values in the result of the query $R_1(x_1, Y)$ may turn out to be a subset of S. In this case, the query $M(x_1, Y, Z)$ can be answered successfully. □

Based on Example 5.5.1, we note that the answerability of a user query is not entirely determined by checking it against the templates computed according to the methods of Sections 5.3 and 5.4. If the query satisfies some template, it is guaranteed to be answerable, otherwise, its answerability depends on the current state of the data in the source views. For instance, the current state of R_1 in Example 5.5.1 may help make the query answerable. Then again, the state of R_1 may be such that the query cannot be answered. Mediators attempting to execute queries that are not guaranteed to be answerable, and determine query answerability in a data-dependent manner at run time, are called *dynamic* mediators.

5.5.1 Conservative and Liberal Templates

Dynamic mediators execute queries that are not guaranteed to be feasible, attempting to answer them in a data-dependent manner. However, as illustrated by Example 5.5.2, it is sometimes possible to determine that a query is infeasible without attempting to execute it.

EXAMPLE 5.5.2 Consider a mediator view $M(X, Y, Z, U)$ defined as a join of two source views $R_1(X, Y)$ and $R_2(Y, Z, U)$. Let R_1 have the single template *bf* and let R_2 have the single template *c[S]fb*, where S is $\{y_1, y_2, y_3\}$. Then, M has the template *bc[S]fb*. This template of M indicates that the query $M(x_1, y_1, Z, u_1)$ is definitely answerable. The query $M(x_1, Y, Z, u_1)$ does not satisfy the template of M. However, it is answerable if the set of Y values at R_1 is a subset of S. The query $M(x_1, Y, z_1, U)$ also does not satisfy the template *bc[S]fb*. We can determine that this query is not going to be answerable because it does not specify a value for U. Blindly trying to execute it in a dynamic mediator results in an expensive way of finding out that the query is infeasible. □

Example 5.5.2 showed that there may be situations in which we can determine that a given query is not going to be answerable irrespective of the state of the data in the sources. It is desirable to be able to specify a set of templates such that if none of them is satisfied by a query, we can ascertain that the query is infeasible, without trying to execute it in vain. Such templates specify an “upper bound” to the set of queries that can be answered

	f	**o**$[S_{21}]$	**b**	**c**$[S_{22}]$	**u**
f	f	f	f	f	f
o$[S_{11}]$	f	f	f	f	f
b	b	b	b	b	b
c$[S_{12}]$	c$[S_{12}]$	c$[S_{12}]$	c$[S_{12}]$	c$[S_{12}]$	c$[S_{12}]$
u	f	f	f	f	f

Table 5.6: Mapping Function for Liberal Join-View Templates.

by a dynamic mediator, while the templates computed in the previous sections form the "lower bound." We call the first kind *liberal* templates and the second kind *conservative* templates.

Each view has a set of conservative templates and a set of liberal templates. Given a query on a view, we ascertain that the query is answerable if it satisfies at least one of the conservative templates of the view, and the query is not answerable if it does not satisfy any of the liberal templates of the view. If a query does not satisfy any conservative templates but satisfies at least one liberal template, then a dynamic mediator executes the query in a data-dependent manner. For instance, in Example 5.5.2, we can specify a conservative template *bc[S]fb* and a liberal template *bffb* for the mediator view M. Based on these templates, we can determine that query $M(x_1, y_1, Z, u_1)$ is guaranteed to be answerable; query $M(x_1, Y, Z, U)$ is guaranteed to be unanswerable; query $M(x_1, Y, Z, u_1)$ may be answerable depending on the state of the data in R_1.

5.5.2 Computing Liberal Templates

Typically, in the case of a source view, the liberal templates of the view are the same as the conservative templates. When computing the templates of derived views at a mediator, the liberal templates tend to diverge from their conservative counterparts. The algorithms of Section 5.4 yield conservative templates for mediator views. That is, we start with conservative templates of base views and obtain conservative templates of derived views.

With small changes to the algorithms of Section 5.4, we can compute the liberal templates. For the selection, projection and union views, we use the same algorithms to compute the liberal templates of derived views starting with the liberal templates of their base views. However, the computation is slightly different in the case of join views.

For a join view, we start with the liberal templates of the base views and compute the corresponding liberal templates of the join view, in a manner that is quite similar to that of

Section 5.4.2. The only difference is the use of a new mapping function that combines the attribute adornments of the base-view templates. The new mapping function is given in Table 5.6. This mapping function is similar to the one used in Section 5.4.2, except in the case of the c adornment in the second base-view template. In this case, when computing the liberal template of the join view, the mediator allows a more flexible adornment because it can try to get the appropriate constant required by the second base view from the result of the query on the first base view. That is, it can optimistically treat the c adornment in the second base-view template as if it is the *f* adornment. For instance, when the first base-view adornment is *f* and the second base-view adornment is c, the liberal adornment for the join-view is *f* (instead of c in the conservative computation of Section 5.4.2). Also, note that when both the base view adornments are c, the resulting c adornment in the liberal template for the join-view has the same constant set as the c adornment in the first base view (instead of the intersection of the constant sets in the two base-view templates).

5.5.3 Content Information and Query Answerability

We have seen how dynamic mediators may be able to answer queries even though the queries do not satisfy the conservative templates. However, in such situations the mediators cannot ascertain the answerability of the queries until run time. Here we briefly explore mediators that use information about the contents of the source views in order to ascertain query answerability statically. There are many interesting issues related to content description of sources and how it affects mediators built on top of them [40, 42, 43].

We note that content descriptions have an important subtle difference from capability descriptions. For instance, in Example 5.5.1, declaring that R_2 has a template *c[S]f*, where S is $\{y_1, y_2, y_3\}$, indicates that queries to R_2 must specify one of these three constants for the Y attribute. The *c[S]f* template of R_2 does not mean that R_2 has no tuple with a Y value that is not in S. In fact, R_2 could very well have a tuple with a Y value y_4, and would refuse to answer the query $R_2(y_4, Z)$. On the other hand, if we were to have a content description of R_2 that declares that the set of Y values at R_2 is S, a mediator can answer the query $R_2(y_4, Z)$ by returning an empty result.

Content descriptions of sources help determine query feasibility at compile time in dynamic mediators. For instance, in Example 5.5.1, if we have a content description of R_1 indicating that the set of Y values at R_1 is a subset of S (the constant set for the Y attribute in the template of R_2), M can have the more flexible template *bff*, instead of the earlier template *bc[S]f*. Note that there is a clear difference between such content-based templates and the liberal templates discussed earlier. The new *bff* template is not really an "upper" bound. If a query satisfies this new template, it is guaranteed to be answerable. In this sense, the *bff* template is a more flexible conservative template, not a liberal template.

5.6 Output Restrictions on Attributes

The computation of mediator templates discussed so far assumes that all the attributes of a view are returned in response to any query on that view. There are situations in which a view may have attributes on which conditions may be specified, but these attributes are not returned in the answer. For example, we can pose a query to Amazon.com [86] by specifying the subject attribute of the desired set of books. At the same time, Amazon does not return the subject attribute when answering queries. For instance, when a user queries Amazon for books written by a specified author, the returned list of books does not mention their subject areas. That is, subject is a query attribute, not an output attribute in Amazon's results. There are also situations in which data sources return different attributes for different queries (see Section 2.2.3).

In order to represent sources that do not return certain attributes, we need to specify explicitly the output restrictions of attributes in the templates of the views exported by the sources. In a template, each attribute should be adorned to reflect its input (query) as well as its output (result) restrictions. To describe the input requirements of an attribute that has no output restriction (i.e., it appears in the result), we use the adornments introduced in Section 5.2: *f*, *o* , *b*, *c* and *u*. To describe the input restrictions of an attribute whose output is suppressed (i.e., it does not appear in the result), we introduce five new adornments: *f'*, *o'*, *b'*, *c'* and *u'*. To illustrate the use of the new adornments, consider a source that exports view $R(X, Y, Z)$ with the requirement that queries on this view must specify X, must not specify Z, and can specify Y optionally. Let the source suppress Y in its output (i.e., X and Z are output, Y is not). We describe the capabilities of the source with a *bf'u* template on R.

The computation of mediator templates in the presence of output restrictions can be undertaken by modified versions of the algorithms in Sections 5.3, 5.4 and 5.5. Only the mapping functions used by the algorithms have to be extended to deal with the new adornments.

	f	**o**$[S_{21}]$	**b**	**c**$[S_{22}]$	**u**	**f'**	**o'**$[S_{23}]$	**b'**	**c'**$[S_{24}]$	**u'**
f	f	o$[S_{21}]$	b	c$[S_{22}]$	u	f	o$[S_{23}]$	b	c$[S_{24}]$	u
o$[S_{11}]$	o$[S_{11}]$	o$[S_{11} \cap S_{21}]$	c$[S_{11}]$	c$[S_{11} \cap S_{22}]$	u	o$[S_{11}]$	o$[S_{11} \cap S_{23}]$	c$[S_{11}]$	c$[S_{11} \cap S_{24}]$	u
b	b	c$[S_{21}]$	b	c$[S_{22}]$	-	b	c$[S_{23}]$	b	c$[S_{24}]$	-
c$[S_{12}]$	c$[S_{12}]$	c$[S_{12} \cap S_{21}]$	c$[S_{12}]$	c$[S_{12} \cap S_{22}]$	-	c$[S_{12}]$	c$[S_{12} \cap S_{23}]$	c$[S_{12}]$	c$[S_{12} \cap S_{24}]$	-
u	u	u	-	-	u	u	u	-	-	u
f'	f	o$[S_{21}]$	b	c$[S_{22}]$	u	f'	o'$[S_{23}]$	b'	c'$[S_{24}]$	u'
o'$[S_{13}]$	oS_{13}]	o$[S_{13} \cap S_{21}]$	c$[S_{13}]$	c$[S_{13} \cap S_{22}]$	u	o'$[S_{13}]$	o'$[S_{13} \cap S_{23}]$	c'$[S_{13}]$	c'$[S_{13} \cap S_{24}]$	u'
b'	b	c$[S_{21}]$	b	c$[S_{22}]$	-	b'	c'$[S_{23}]$	b'	c'$[S_{24}]$	-
c'$[S_{14}]$	c$[S_{14}]$	c$[S_{14} \cap S_{21}]$	c$[S_{14}]$	c$[S_{14} \cap S_{22}]$	-	c'$[S_{14}]$	c'$[S_{14} \cap S_{23}]$	c'$[S_{14}]$	c'$[S_{14} \cap S_{24}]$	-
u'	u	u	-	-	u	u'	u'	-	-	u'

Table 5.7: Simple Union and Join with Output Restrictions.

5.6.1 Template Computation for Simple Mediators

Here we present the new mapping functions for the algorithms of Section 5.3, which compute templates for simple mediators.

Union Views

The new mapping function for computing union-view templates in simple mediators is given in Table 5.7. Note that the new mapping function considers all ten attribute adornments (as opposed to the five considered by Table 5.1). However, since the simple mediators do not perform any postprocessing, the entries in the new mapping function are quite similar to those of Table 5.1.

The mediator can output a union-view attribute if both base views output the attribute. The mediator cannot output a union-view attribute if neither base view outputs the attribute. As shown in Table 5.7, we adopt the policy that if an attribute is output by only one of the base views, the mediator outputs the attribute in the union view. When taking the union of the results of the two base-view queries, the mediator can introduce *NULL* values for the missing attribute values. Such a policy is adopted in practice by Web mediators like the Yahoo shopping guide for books [97].

Join Views

As in Section 5.3, the computation of join-view templates uses the same mapping function as the computation of union-view templates. Therefore, the new mapping function for computing join-view templates in simple mediators in the presence of output restrictions is

	f	**o**[S_{21}]	**b**	**c**[S_{22}]	**u**	**f'**	**o'**[S_{23}]	**b'**	**c'**[S_{24}]	**u'**
f	f	f	b	c[S_{22}]	f	f	o[S_{23}]	b	c[S_{24}]	u
o[S_{11}]	f	f	b	c[S_{22}]	f	f	o[S_{23}]	b	c[S_{24}]	u
b	b	b	b	c[S_{22}]	b	b	c[S_{23}]	b	c[S_{24}]	-
c[S_{12}]	c[S_{12}]	c[S_{12}]	c[S_{12}]	c[$S_{12} \cap S_{22}$]	c[S_{12}]	c[S_{12}]	c[$S_{12} \cap S_{23}$]	c[S_{12}]	c[$S_{12} \cap S_{24}$]	-
u	f	f	b	c[S_{22}]	f	f	o[S_{23}]	b	c[S_{24}]	u
f'	f	f	b	c[S_{22}]	f	f'	o'[S_{23}]	b	c[S_{24}]	u'
o'[S_{13}]	o[S_{13}]	o[S_{13}]	c[S_{13}]	c[$S_{13} \cap S_{22}$]	o[S_{13}]	o'[S_{13}]	o'[$S_{13} \cap S_{23}$]	c[S_{13}]	c[$S_{13} \cap S_{24}$]	u'
b'	b	b	b	c[S_{22}]	b	b	c[S_{23}]	b	c[S_{24}]	-
c'[S_{14}]	c[S_{14}]	c[S_{14}]	c[S_{14}]	c[$S_{14} \cap S_{22}$]	c[S_{14}]	c[S_{14}]	c[$S_{14} \cap S_{23}$]	c[S_{14}]	c[$S_{14} \cap S_{24}$]	-
u'	u	u	-	-	u	u'	u'	-	-	u'

Table 5.8: Union with Postprocessing and Output Restrictions.

also given by Table 5.7.

Selection Views

As in Section 5.3.3, each base-view template yields a corresponding selection-view template with the adornments copied over. This direct mapping is an obvious extension of the one in Section 5.3.3, as it copies over the new adornments as well as the old adornments.

Projection Views

As in Section 5.3.3, we attempt to create a projection-view template from each base-view template. Whenever a base-view template has a *b*, *c*, *b'* or *c'* adornment for a hidden attribute, we do not derive a projection-view template. Otherwise, we copy over the adornment for each projected attribute from the base-view templates to the corresponding projection-view templates. That is, the new mapping function for computing projection-view templates in simple mediators is a direct extension of the one in Section 5.3.4.

5.6.2 Template Computation for Complex Mediators

Here we present the new mapping functions for the algorithms of Section 5.4, which compute templates for mediators that perform postprocessing operations and bind joins.

Union Views

The new mapping function for computing union-view templates in postprocessing mediators is given in Table 5.8. Notice that in the first five rows and the first five columns (the top-left quadrant) of Table 5.8, both base views output the attribute in their results. Therefore, the mediator can perform postprocessing as in Section 5.4 and so the entries are identical to those of Table 5.2. The entries in the last five rows and the last five columns (the bottom-right quadrant) of Table 5.8 are similar to Table 5.1 because both base views are suppressing the attribute, and so the mediator cannot perform any postprocessing on this attribute. In the rest of Table 5.8, the mediator can postprocess on the attribute returned from one of the base views. Consequently, the entries are less flexible than the top-left quadrant and are more flexible than the bottom-right quadrant. For instance, the combination of *u* and *o* yields *f* (top-left quadrant); the combination of *u'* and *o'* yields *u'* (bottom-right quadrant); the combination of *u* and *o'* yields *o* (top-right quadrant); and the combination of *u'* and *o* yields *u* (bottom-left quadrant).

Regarding the output of attributes in a union view, we adopt the policy that the mediator can output an attribute if at least one of the two base views outputs the attribute. We observe that a "smart" mediator can alleviate the output restrictions on attributes that are already specified in the queries. For instance, a *b'* adornment for an attribute in a template indicates that a value must be specified for the attribute in a feasible query, and that the attribute will not be output in the result of the query. In this case, the mediator can simply copy the input value from the query to the query result, thus allowing the attribute to be present in the output. Hence, the *b'* adornment can be treated in the same way as the less restrictive *b* adornment. Similarly, the *c'* adornment may be treated as the *c* adornment. Notice that the cases of *f'* and *o'* where an input value is specified in query can also lead to the attribute being present in the output. However, the mediator cannot guarantee that the attribute will be present when the adornment is *f'* or *o'*. That is, it cannot convert *f'* to *f* and *o'* to *o*. The entries in Table 5.8 reflect the more flexible adornments we are discussing here (i.e., *b'* becomes *b* and *c'* becomes *c*).

Join Views

Table 5.9 presents the new mapping function for computing the adornment of a join attribute in a join sequence of two base-views with single templates. The *f'* and the *o'* adornments of the base-view templates are treated in a similar fashion to the *b'* and the *c'* adornments. The reason is that when a base-view query does not output the attribute, the attribute cannot be left unspecified in the base-view query because local joins are not possible due to the absence of the join attribute in the result of the base-view query. Note that if the

	f	**o**[S_{21}]	**b**	**c**[S_{22}]	**u**	**f'**	**o'**[S_{23}]	**b'**	**c'**[S_{24}]	**u'**
f	f	f	f	c[S_{22}]	f	f	c[S_{23}]	f	c[S_{24}]	-
o[S_{11}]	f	f	f	c[S_{22}]	f	f	c[S_{23}]	f	c[S_{24}]	-
b	b	b	b	c[S_{22}]	b	b	c[S_{23}]	b	c[S_{24}]	-
c[S_{12}]	c[S_{12}]	c[S_{12}]	c[S_{12}]	c[$S_{12} \cap S_{22}$]	c[S_{12}]	c[S_{12}]	c[$S_{12} \cap S_{23}$]	c[S_{12}]	c[$S_{12} \cap S_{24}$]	-
u	f	f	f	c[S_{22}]	f	f	c[S_{23}]	f	c[S_{24}]	-
f'	b	b	b	c[S_{22}]	b	b	c[S_{23}]	b	c[S_{24}]	-
o'[S_{13}]	c[S_{13}]	c[S_{13}]	c[S_{13}]	c[$S_{13} \cap S_{22}$]	c[S_{13}]	c[S_{13}]	c[$S_{13} \cap S_{23}$]	c[S_{13}]	c[$S_{13} \cap S_{24}$]	-
b'	b	b	b	c[S_{22}]	b	b	c[S_{23}]	b	c[S_{24}]	-
c'[S_{14}]	c[S_{14}]	c[S_{14}]	c[S_{14}]	c[$S_{14} \cap S_{22}$]	c[S_{14}]	c[S_{14}]	c[$S_{14} \cap S_{23}$]	c[S_{14}]	c[$S_{14} \cap S_{24}$]	-
u'	-	-	-	-	-	-	-	-	-	-

Table 5.9: Join with Postprocessing and Output Restrictions.

Base-View Adornment	Selection-Attribute Adornment	Nonselection-Attribute Adornment
f	f	f
o[S_1]	f	f
b	f or b	b
c[S_2]	f or c[S_2]	c[S_2]
u	f	f
f'	f or f'	f'
o'[S_3]	f or o'[S_3]	o'[S_3]
b'	f or b	b
c'[S_4]	f or c[S_4]	c[S_4]
u'	-	u'

Table 5.10: Selection with Postprocessing and Output Restrictions.

join attribute has the *u'* adornment in either base-view, no template for the join sequence is derived because the mediator cannot perform the join operation. Local join is ruled out because the attribute is not returned in the base-view query result, while bind join is ruled out because the attribute cannot be specified in the base-view query. Also, the *b* adornment of the second base view is treated as the *f* adornment only when the first base view query returns the attribute.

Selection and Projection Views

The new mapping function for computing selection-view templates in a postprocessing mediator is given by Table 5.10. The top five rows of this table are identical to Table 5.4 because the mediator can perform all necessary postprocessing on the attribute that is output in the results of the base-view query.

In the bottom five rows, if the attribute does not participate in the selection predicate, its adornment is similar to the base-view adornment because no postprocessing of the attribute is undertaken by the mediator. If the attribute participates in the selection predicate, we may not be able to derive a selection-view template because the mediator cannot evaluate the selection predicate involving the invisible attribute when postprocessing the results of the base-view query. There are cases in which the mediator can avoid the need to perform postprocessing steps on the invisible attribute, based on its ability to infer required values for the invisible attribute from the selection predicate. If the mediator can infer a specific value that an invisible attribute must satisfy, it can input this value in the query to the base-view. Then, the mediator need not postprocess the result of the base-view query to check for the satisfaction of the required value for the invisible attribute. Note that if the base-view template has the *u'* adornment for an invisible attribute that participates in the selection predicate, the mediator will not be able to process the selection predicate and so it will not have a corresponding selection-view template.

In the case of *f'* and *b'* adornments of the base-view template, if the mediator can infer a required value for the attribute, then it can use this value to bind the base-view query and also allow the selection-view query to specify any value for that attribute. If the selection-view query specifies a value different from the value inferred from the predicate, the mediator can simply answer the selection-view query by returning an empty result. If the selection-view query specifies the same value, it can be used to bind the base-view query. If the selection-view query does not specify any value, the mediator can bind the base-view query with the value inferred from the selection predicate. Accordingly, for these two base-view adornments, we have the *f* adornment for the selection-view. In the case of the *o'* and *c'* base-view adornments, if the inferred value is one of the menu constants, the selection-view can have the *f* adornment. Finally, as before, we assume that the mediator is smart enough to convert *b'* to *b* and *c'* to *c*.

The new mapping function for a projection view is given in Table 5.11. Note that the entries in the top five rows are identical to the entries in Table 5.5 because the mediator can perform postprocessing on the projected attribute. For the entries in the bottom five rows, the mediator cannot perform any postprocessing. So, it merely copies over the adornment from the base-view template. As before, we do not derive a template for the projection view, if the base view has an adornment of *b*, *c*, *b'* or *c'* for any of the hidden attributes.

Base-View Adornment	Projection-View Adornment
f	f
o[S_1]	f
b	b
c[S_2]	c[S_2]
u	f
f'	f'
o'[S_3]	o'[S_3]
b'	b
c'[S_4]	c[S_4]
u'	u'

Table 5.11: Projection with Postprocessing and Output Restrictions.

	f	**o**[S_{21}]	**b**	**c**[S_{22}]	**u**	**f'**	**o'**[S_{23}]	**b'**	**c'**[S_{24}]	**u'**
f	f	f	f	f	f	f	f	f	f	-
o[S_{11}]	f	f	f	f	f	f	f	f	f	-
b	b	b	b	b	b	b	b	b	b	-
c[S_{12}]	c[S_{12}]	c[S_{12}]	c[S_{12}]	c[S_{12}]	c[S_{12}]	c[S_{12}]	c[S_{12}]	c[S_{12}]	c[S_{12}]	-
u	f	f	f	f	f	f	f	f	f	-
f'	b	b	b	b	b	b	c[S_{23}]	b	b	-
o'[S_{13}]	c[S_{13}]	c[S_{13}]	c[S_{13}]	c[S_{13}]	c[S_{13}]	c[S_{13}]	c[S_{13}]	c[S_{13}]	c[S_{13}]	-
b'	b	b	b	b	b	b	b	b	b	-
c'[S_{14}]	c[S_{14}]	c[S_{14}]	c[S_{14}]	c[S_{14}]	c[S_{14}]	c[S_{14}]	c[S_{14}]	c[S_{14}]	c[S_{14}]	-
u'	-	-	-	-	-	-	-	-	-	-

Table 5.12: Liberal Templates of Join Views with Output Restrictions.

Also, as before, we convert the *b'* to a *b* and the *c'* to a *c* adornment for the mediator view.

5.6.3 Template Computation for Dynamic Mediators

Here we present the new mapping functions for the dynamic mediators of Section 5.5. Note that the dynamic mediators use the same algorithm as in Section 5.4 for computing the conservative templates of all its views. They also use the same algorithm for computing the liberal templates of union views, selection views and projection views. Therefore, the new mapping functions for computing all the templates except the liberal templates of join views are the same as those of Section 5.6.2. Here we present the new mapping function for computing liberal templates of join views.

The new mapping function to compute the liberal adornments of join views (Table 5.12) is quite similar to the one used in computing the conservative templates for join views with output restrictions (Table 5.9). The essential difference is that when we compute the liberal adornments, we optimistically assume that the bindings returned for the join attribute from the query on the first base view are in the menus of choices for the attribute when the second base-view adornments are *o*, *o'*, *c* and *c'*. For instance, when the first base-view adornment is *f* and the second base-view adornment is *c'*, Table 5.12 indicates the resulting adornment of *f* (instead of *c* in Table 5.9). We also have less constrained *c* and *o* adornments of the attribute in the join-view template when it has adornments with menus in both the base-view template.

5.7 A Case Study

To verify that our capability-computation framework makes sense in practice, we explored the Web, where many limited-capability sources are found. In particular, we wanted to determine if the adornments we developed in our framework were adequate in describing the query capabilities of sources and mediators. We also wanted to know how many templates were typically required to describe sources and mediators. Capability-based query processing could become unwieldy if large numbers of templates were required. Therefore, it is important to check if in the case of representative sources and mediators there would be an explosion of templates.

We started by examining a mediator for two bookstores: Amazon.com [85] and BarnesAndNoble.com [90]. Amazon had the following three query forms [86, 87, 88]:

- Form 1: At least one of author, title and subject attributes must be specified.
- Form 2: The ISBN attribute must be specified.
- Form 3: At least one of keywords, publisher and publication date attributes must be specified.

The results of queries to Amazon included the following attributes: author, title, ISBN, publisher, date, format (paperback or hardcover), price and shipinfo (availability and shipping information). We define the following view to describe the content of this Web source:

$$Book_1(author, title, subject, KW, ISBN, pub, date, format, price, ship).$$

The query capabilities of Amazon are described by the templates in Table 5.13.[4] The capabilities offered by each query form are captured by one or more templates. In Table 5.13, the first three templates capture Form 1, the fourth template captures Form 2, while the last three templates capture Form 3.

author	**title**	**subject**	**KW**	**ISBN**	**pub**	**date**	**format**	**price**	**ship**
b	f	f'	u'	u	u	u	u	u	u
f	b	f'	u'	u	u	u	u	u	u
f	f	b'	u'	u	u	u	u	u	u
u	u	u'	u'	b	u	u	u	u	u
u	u	u'	f'	u	b	f	u	u	u
u	u	u'	b'	u	f	f	u	u	u
u	u	u'	f'	u	f	b	u	u	u

Table 5.13: Templates of Amazon.

BarnesAndNoble had two query forms [91]. The first one supported search by ISBN. The second form allowed search by specifying at least one of author, keywords and title attributes. In addition, through the second form, one could specify price limit, publishing format, subject area, and recommended age range. The output attribute set of BarnesAndNoble was the same as that of Amazon. We define the following schema for BarnesAndNoble.

$$Book_2(author, title, subject, KW, ISBN, pub, date, price, format, ship, age).$$

The capabilities of BarnesAndNoble are described by the templates in Table 5.14. The first three templates describe the capabilities of the form through which we could search by author, keywords and title, while the fourth template describes the form with the ISBN attribute. The price, format and subject attribute specifications had menus to choose from. These attributes could also be left unspecified. Therefore, they have the o adornment with their respective menus. For instance, the format attribute in the first template has the o adornment with a menu of options: "Hardcover," "Paperback," "Books on Cassette or CD," and "Large Print." For the sake of simplicity, we do not list the menus of options for

[4]Some of the conditions specified on certain attributes may not be *equality based*. For instance, the value specified for the title condition may translate to checking if the returned books have titles that contain the specified word or words, instead of having exactly the matching titles. For the discussion here, we consider the special semantics and predicates applied for such conditions in the same way as the equality conditions. Related literature (e.g., [12]) throws more light on the issues surrounding such special semantics and predicates and their distinction from the equality predicate.

attribute adornments in the tables of this section.

author	title	subject	KW	ISBN	pub	date	format	price	ship	age
f	b	o'	f'	u	u	u	o	o	u	o'
b	f	o'	f'	u	u	u	o	o	u	o'
f	f	o'	b'	u	u	u	o	o	u	o'
u	u	u'	u'	b	u	u	u	u	u	u'

Table 5.14: Templates of BarnesAndNoble.

Suppose we now build a mediator on these two Web bookstores. The mediator defines the following union view over the two source views:

$$Book(author, title, subject, KW, ISBN, pub, date, price, format, ship, age).$$

author	title	subject	KW	ISBN	pub	date	format	price	ship	age
b	b	o'	u'	f	f	f	f	f	f	u'
b	f	o'	u'	f	f	f	f	f	f	u'
b	f	u'	u'	b	f	f	f	f	f	u'
f	b	o'	u'	f	f	f	f	f	f	u'
f	b	u'	u'	b	f	f	f	f	f	u'
f	b	c'	u'	f	f	f	f	f	f	u'
b	f	c'	u'	f	f	f	f	f	f	u'
f	f	u'	u'	b	f	f	f	f	f	u'
f	b	u'	f'	f	b	f	f	f	f	u'
b	f	u'	f'	f	b	f	f	f	f	u'
f	f	u'	b'	f	b	f	f	f	f	u'
f	f	u'	u'	b	b	f	f	f	f	u'
f	b	u'	b'	f	f	f	f	f	f	u'
b	f	u'	b'	f	f	f	f	f	f	u'
f	f	u'	b'	f	f	f	f	f	f	u'
f	b	u'	f'	f	f	b	f	f	f	u'
b	f	u'	f'	f	f	b	f	f	f	u'
f	f	u'	b'	f	f	b	f	f	f	u'
f	f	u'	u'	b	f	b	f	f	f	u'

Table 5.15: Mediator Templates before Minimization.

In order to compute the templates that describe the set of feasible queries on the union view, let us assume that the mediator employs postprocessing techniques. We ran our algorithm to compute mediator templates (see Section 5.6.2) and obtained 19 templates for the mediator view. These templates are shown in Table 5.15. The algorithm actually considered 28 (4×7) pairs of source-view templates and successfully generated union-view templates in the case of 23 pairs. However, the list of the resulting union templates has duplicates. After removing the duplicates, we ended up with 19 templates.

author	**title**	**subject**	**KW**	**ISBN**	**pub**	**date**	**format**	**price**	**ship**	**age**
f	f	u'	u'	b	f	f	f	f	f	u'
f	f	u'	b'	f	f	f	f	f	f	u'
b	f	o'	u'	f	f	f	f	f	f	u'
f	b	o'	u'	f	f	f	f	f	f	u'
b	f	u'	f'	f	b	f	f	f	f	u'
f	b	u'	f'	f	b	f	f	f	f	u'
f	b	u'	f'	f	f	b	f	f	f	u'
b	f	u'	f'	f	f	b	f	f	f	u'

Table 5.16: Mediator Templates after Minimization.

The resulting set of 19 templates can be reduced to a set of 8 mediator templates (see Table 5.16), using minimization techniques discussed in Chapter 6. From the set of 8 templates, 4 query forms can be derived for the mediator. Corresponding to the first template in Table 5.16, the bookstore mediator has a query form that requires the specification of the ISBN attribute. Corresponding to the second template, we create a query form that requires the keywords to be specified. Next, we create a single query form that corresponds to the third and fourth templates. This query form requires that at least one of author and title attributes must be specified along with an optional specification of the subject field from a menu of choices. Finally, the last four templates are combined into a single query form that requires that at least one of author and title attributes and at least one of publisher and date attributes must be specified. Note that, the theoretical maximum number of forms for our bookstore is 28 (because each combination of the base-view templates could have resulted in a mediator template, and each mediator template could end up as a separate query form). The fact that in our case study we obtained 4 query forms for the mediator suggests that we may be able to compute manageable sets of query forms for mediators on Web sources.

Our case study indicates that the capability-description languages used by many contemporary mediation systems are not suitable to describe sources on the Web. In particular, the lack of proper support for menus of constants and output restrictions in these languages is a big problem. In contrast, our capability-description framework (see Section 5.2) is well suited to describe the capabilities of Web sources and mediators.

We studied how query forms can be translated into templates and vice versa. In some cases, a query form can be described by a single template. In some other cases, a query form may be represented by a set of templates. The most common situation where we need a set of templates to describe a form arises when the form specifies that at least one of a set of attributes must be filled in. We also studied the process of deriving query forms from templates. It is very easy to derive a query for each template. For instance, in our bookstore mediator we could generate 8 query forms, one for each template. However, there are opportunities to capture the capabilities described by a set of templates through a single query form. For instance, we derived a query form for the bookstore mediator that covered four templates. The process of deriving query forms from templates involves steps that retrace the process of deriving templates from query forms. In general, we note that it is usually easier to derive templates from query forms than vice versa.

In our case study, we encountered interesting ways in which pop-up menus are used in many Web forms. Some menus have a "default" option of "all" or "any." For example, the menu for the subject attribute in the search form of BarnesAndNoble had a default constant "all areas." In our capability-description framework, we can capture such an attribute restriction with either a c adornment or an o adornment. In the former case, we can list all the menu options, including the default one, in the constant set. If we use the o adornment, we can leave out the default option and list all the other options. We adopted the latter approach in our case study.

In the course of our experiments, we noticed that many Web sources change their query forms frequently. In our experience, sources change their Web forms many times in a short period of time (a few times a year). Building mediators on such evolving sources poses special challenges. In particular, manual generation of query forms for mediators on Web sources becomes very difficult because whenever sources change, mediator capabilities have to be re-assessed. Automatic computation of mediator capabilities based on the techniques presented in this chapter can be very helpful to mediation systems involving frequently changing sources.

We also observed that there are many Web sources that provide overlapping and related information. For example, there are many travel agencies on the Web. Mediators on the Web have to expect that large numbers of sources are integrated by mediator views. Once again systematic computation of mediator capabilities in this situation is essential.

5.8 Related Work

In this section, we briefly discuss the approaches taken by contemporary mediation systems to deal with query-capability limitations of the data sources they integrate. TSIMMIS [29] uses *query templates* to describe source capabilities. In a template, each attribute can be adorned as *bound*, *free* or bound to a single fixed *constant*. The mediator defines integrated views on the source views using the MSL language [55]. An MSL view definition specifies the content of the view, but does not indicate which queries can be supported using the view. When a user submits a query, the mediator translates the query into a logical plan by expanding the views specified in the query, and, if possible, generates a *feasible* physical plan based on the source templates [44]. At the time of submitting a query, the user does not know if the query is actually executable, i.e., if a feasible plan will be found. The TSIMMIS mediator does not describe its capabilities to the user, so the user has to submit queries in a trial-and-error scenario.

The Information Manifold [40] uses *capability records* to describe source capabilities. Each source has a capability record that indicates the input attributes of a source view, its output attributes, the smallest number of input attributes that must be specified in a query to the source, and the attributes on which the source can apply selections of the form α *op* c, where c is a constant and $op \in \{\leq, <, \neq, =\}$. Capability descriptions in the Information Manifold are limited to data sources. As in TSIMMIS, an Information Manifold mediator does not compute and export its capabilities to users, and other mediators using it as a source.

Garlic [9, 28, 57] is a mediation system that integrates large-scale multimedia information systems. The metadata in Garlic does not include the query-processing capabilities of the individual sources, so the query processor at the mediator has no direct knowledge about the capabilities of the sources being integrated. In order to make sure that the plan generated by the mediator is feasible, the sources participate in the mediator's query-planning process. That is, each source encapsulates the knowledge of its own capabilities and lets the mediator know if a given query will be answerable by the source. As with TSIMMIS and Information Manifold, Garlic mediators do not compute capability descriptions. That is, users posing queries to a Garlic mediator have no idea about the set of answerable queries.

Other integration systems, like DISCO [71], SIMS [4] and Araneus [47], use different languages to describe source capabilities. Again, mediators in these systems do not compute and export their capabilities.

The NAIL! system [51] studies the problem of generating query plans for integrated views in the presence of source-capability restrictions. However, NAIL! only considers a limited set of restrictions expressed as attribute adornments, *free* and *bound*. Also, NAIL! does not precompute the adornments for integrated views. In contrast, we consider a richer set of attribute adornments and precompute feasible adornments of integrated views. The LDL system [14, 74] uses the notion of query forms. Given a set of query forms, it precomputes skeleton query plans for them. When a query is submitted, if it fits one of the precomputed query forms, the LDL system can instantiate the corresponding skeleton plan to execute the query. The results we develop here complement their work by identifying the set of mediator query forms for which skeleton plans may be precomputed.

Related research has explored heterogeneous-database schema integration and data translation [1, 8, 49], as well as capability-based query optimization [20, 57, 80]. The techniques we developed to compute mediator capabilities complement those efforts.

5.9 Chapter Summary

In data-integration systems, it is important to describe the capabilities of mediators so that they can be used as easily (by end users as well as other applications) as base sources are. Many contemporary integration systems do not compute and exported mediator capabilities, thus making it hard for them to be useful in scalable applications involving networks of mediators and sources. In some situations, mediator capabilities are manually computed and specified. Manual computation is error prone and becomes unwieldy when dealing with large numbers of evolving sources whose query capabilities change frequently.

In this chapter, we provided the machinery for automatically computing the capabilities of mediators based on the capabilities of the sources they integrate. We proposed a capability-description framework in which a rich set of attribute adornments are used to describe a variety of query-processing limitations of sources and mediators. We discussed various classes of mediators based on the query processing techniques they employ, and presented algorithms for the computation of mediator capabilities for the various classes. We conducted experiments using Web sources and studied issues surrounding the adequacy of our capability-description framework and the effectiveness of our algorithms for computing mediator capabilities.

Chapter 6

Minimization of Template Sets

6.1 Overview

In the previous chapter, we presented a framework in which the query capabilities of data sources can be expressed elegantly. We have also developed algorithms to compute mediator templates starting from the query templates for the data sources they mediate. The number of templates computed for a mediator view can be exponential in terms of the number of templates of the sources participating in the view. The capability description of a mediator, expressed as its query templates, may be so large that determining if a given query is answerable becomes very difficult. Users trying to pose queries to the mediator may not be able to easily figure out what queries are feasible.

Through a process of minimizing template sets, we can compute concise capability descriptions of mediators. In this chapter, we develop techniques to solve the template-set minimization problem. Minimizing template sets is also a key step in the process of designing query interfaces to data sources, as described below.

6.1.1 Designing Source Interfaces

One approach to designing the set of templates for a data source is to start from the query requirements specified by the customer needs. Based on the requirements, we can construct query templates and then combine them to arrive at the interface design for the data source. This process is illustrated by the following example.

EXAMPLE 6.1.1 Consider a Web bookstore that exports the view *BOOKS(author, title, format)*. Customer requirements are gathered by interviewing potential users and determining the queries they are most likely to submit. Each prototypical query leads to one template that should be offered by the source. For example, if users want to search for books by only giving the *author* name, we add the template *ubu*.

Continuing in this manner, suppose that the customer needs lead to the following seven templates: { *buu*, *ubu*, *bbu*, *uub*, *ubb*, *bub*, *bbb* }. We could then meet these needs with seven search forms, one for each template. However, note that all the expected customer needs can also be satisfied with just three templates: $\{bff, fbf, ffb\}$. The first template bff allows all queries permitted by the four templates *buu*, *bbu*, *bub* and *bbb*. The second new template fbf also covers four templates in the original design, but two of them are the same as those covered by the bff template. Finally, the new ffb template also covers four templates, only one of which is not covered by the previous two new templates, namely *uub*.

The data source can support either design for the *BOOKS* interface, the one with the seven templates or the one with the three templates. However, the three-template design is more compact, easier to implement, and easier to maintain. Thus, we believe it is important to minimize template sets to arrive at compact interface descriptions. The algorithms we present in this chapter perform such minimization in an efficient fashion. □

Example 6.1.1 illustrates a "requirements-based" interface design, where we start with user requirements and develop a concise set of templates that specify the query interface supported by the data source. The interface design can also be driven in a "capabilities-based" fashion, starting with the interfaces and capabilities that the underlying data engines can provide. For example, as in the previous chapter, we can compute the set of templates for a mediator, starting with the sets of templates of the various sources it is built upon.

6.1.2 Concise Source Interfaces

Given a set of templates, why is it important to find a smaller set with the same capabilities? There are several reasons, including:

- *User Satisfaction.* A customer wishing to ask a query needs to figure out which search form to use. Obviously, having more than a handful of forms may lead to customer aggravation and consequent loss of business.

- *Query Optimization.* A mediator or application that queries a Web source must develop a good execution plan. Each template offered by the source represents additional potential queries that must be considered in developing a good plan. Thus, the more succinctly the available queries can be described, the more efficient the mediator's query-optimization process will be.

- *Maintenance.* Each form supported by the source must be maintained as software and system evolve (e.g., the form must present the latest graphics, must support the latest browsers). The more forms the source supports, the larger the expense for constructing the corresponding Web pages and upgrading them on a continual basis.

In some contexts, template-set minimization is performed manually and in an ad-hoc fashion. Given a set of templates that describe expected user queries, a human being studies the templates and designs forms that can cover all the expected queries. Unfortunately, manual solutions suffer from three main drawbacks. First, they are error prone (particularly when many attributes are involved and the adornments have menus of choices associated with them). An erroneous interface description can lead to customer aggravation, loss of business, inefficient query planning and execution. Second, manual solutions do not scale well with the increasing complexity of rich interfaces. Third, the interface requirements change often over the life of a Web source. It is not unusual for a service to change significantly the set of answerable queries every few weeks. Incrementally evolving the interfaces by hand is clearly undesirable.

In this chapter, we present algorithms for *automatically* computing concise representations for sets of query templates. Notice that "finding concise representations" may involve identifying and eliminating templates that are redundant in a given set of templates. It may also involve "coalescing" two or more templates in a set into a more powerful one. Or it may involve finding a new set of templates that is smaller and as a whole covers the original set. In particular, this chapter focuses on achieving the following goals:

- Defining the template-set minimization problem (TSMP), as well as related problems like template subsumption, template coalescing, and template redundancy.

- Studying the complexity of the problems, showing in particular that TSMP and the template redundancy problem are *intractable* in general, while the subsumption and coalescing problems can be solved efficiently (in polynomial time).

- Developing simple efficient algorithms for detecting template subsumption and for coalescing templates to form more powerful templates.
- Identifying many practical scenarios of interface design for Web sources that can be mapped to well-defined classes of instances of the TSMP problem, and developing efficient algorithms that yield minimum template sets for these classes.
- Designing approximation algorithms for the general class of arbitrarily complex templates, i.e., the general case of the TSMP problem.

6.2 The Template-Set Minimization Problem

A template indicates a set of queries answerable by the source. We call this set of queries the set of *supported queries* of the template. We define the set of supported queries of a set of templates as the union of the sets of supported queries of each template in the set. A template set is *equivalent* to another template set if they both have the same set of supported queries.

The main problem we are studying in this chapter, which is at the core of designing query interfaces, is formally defined as follows:

Template-Set-Minimization Problem (TSMP). Given a template set T, what is the smallest set of templates M such that M and T are equivalent?

Note that in TSMP, the smallest equivalent set is not restricted to be a subset of the given set of templates (i.e., M does not have to be a subset of T).

The following theorem establishes the complexity of TSMP.

Theorem 6.2.1 *TSMP is intractable.* □

Proof: We argue that TSMP is intractable by showing that a polynomial-time algorithm for it will lead to a polynomial-time algorithm for the well-known $\mathcal{NP}$-complete problem CNF-SAT [21]. Since the CNF-SAT problem is established as an intractable problem, we conclude that TSMP is also intractable.

Given an instance of the CNF-SAT problem, we create a template corresponding to each clause in the CNF expression. Specifically, we define an attribute corresponding to each variable in the CNF expression. The templates specify adornments over this attribute set. When constructing a template corresponding to a clause, the adornment of an attribute is f if the corresponding variable does not appear in the clause, the adornment is b if the variable appears as a positive literal and the adornment is u if the variable appears as a negative literal. Thus, we construct a set of templates T corresponding to the set of clauses in the CNF expression. We now argue that the CNF expression is satisfiable if and only if T cannot be minimized to a singleton set of templates containing the template with f adornment for all attributes.

If T cannot be minimized to $\{fff...f\}$, there must exist a query Q that is not supported by T. Based on Q, let us construct an assignment A such that A assigns 1 to variables whose corresponding attributes are not specified by Q and A assigns 0 to variables whose corresponding attributes are specified by Q. Now, consider a clause c_i in the given CNF expression. Since Q is not supported by the corresponding template t_i, it must be the case that t_i has a u adornment for an attribute that Q specifies or t_i has a b adornment for an attribute that Q does not specify. In the former case, c_i has a negative literal whose variable is assigned a value of 0 in A. In the latter case, c_i has a positive literal whose variable is assigned a value of 1 in A. In other words, A satisfies c_i. Thus, we conclude that if T cannot be minimized to $\{fff...f\}$, the CNF expression is satisfiable.

If the CNF expression is satisfiable, there must be a satisfying assignment A'. Based on A', let us construct a query Q' such that Q' specifies a value for all the attributes whose corresponding variables are assigned a value of 0 in A' and Q' leaves all other attributes unspecified. Now, consider a template t_j in T. Since A' satisfies the corresponding clause c_j, it must be the case that c_j has a positive literal whose variable is assigned a value of 1 in A' or c_j has a negative literal whose variable is assigned a value of 0 in A'. In the former case, Q' leaves unspecified an attribute that has the b adornment in t_j. In the latter case, Q' specifies an attribute that has the u adornment in t_j. That is, Q' is not supported by t_j. Thus, we see that Q' is not supported by any of the templates in T and so T cannot be equivalent to $\{fff...f\}$. In other words, if the CNF expression is satisfiable, T cannot be minimized to $\{fff...f\}$.

Thus, we have shown that a polynomial-time algorithm for TSMP will lead to polynomial-time algorithm for the CNF-SAT problem. Since the CNF-SAT problem is known to be intractable, we conclude that TSMP is also intractable. ∎

Given that TSMP is intractable, the next logical step is to investigate if it remains intractable for special cases arising from restrictions on the kinds of templates involved. One particularly well-known class of templates to consider is the one that does not use menus of choices to restrict the constants that must be specified. That is, we consider templates that use only the b, f and the u adornment. As shown in the proof of Theorem 6.2.1, for this restricted class of templates TSMP remains intractable. However, it turns out that for some other restricted classes of templates, TSMP can be solved efficiently. Moreover, these "good" classes happen to be used often in practice and in many query-processing systems dealing with Web sources. In the next section, we develop an efficient algorithm to solve

	f	**o**$[S_{21}]$	**b**	**c**$[S_{22}]$	**u**
f	yes	no	no	no	no
o$[S_{11}]$	yes	yes if $S_{11} \subseteq S_{21}$	no	no	no
b	yes	no	yes	no	no
c$[S_{12}]$	yes	yes if $S_{12} \subseteq S_{21}$	yes	yes if $S_{12} \subseteq S_{22}$	no
u	yes	yes	no	no	yes

Table 6.1: Adornment Restrictiveness Checking.

TSMP for these restricted classes of templates.

6.3 Template Subsumption

One approach to obtaining concise source designs is to look for redundant templates in a given design. Each template has a set of supported queries. If all the queries supported by a template are also supported by another template, then the first template can be eliminated from any set that contains the second one. For instance, all the queries supported by the template $ubfb$ are also supported by the template $fbff$. So, if we are given the design $\{ubfb, fbff\}$ we can come up with the more concise design $\{fbff\}$.

We formalize the notion of identifying redundant templates as described above. We say that a template t subsumes another template t' if every query supported by t' is also supported by t. Based on this notion of template subsumption, we develop a very efficient algorithm for minimizing query interfaces. It turns out that this simple algorithm can yield optimal designs in many contexts involving restricted classes of templates.

We need to first construct an algorithm to test for template subsumption. We can ascertain if a template t subsumes another template t', based on the notion of the degree of restrictiveness of the attribute adornments, which in turn is specified by Table 6.1. The adornment on the left side of the table is at least as restrictive as the adornment on the top of the table if the corresponding entry in the table says "yes." For instance, b is more restrictive than f and c is more restrictive than b.

Intuitively, a template that has more restrictive adornments is subsumed by a template that has less restrictive adornments. This intuition is formalized in the following lemma.

Lemma 6.3.1 *A template subsumes another if and only if for each attribute, the adornment in the latter is at least as restrictive as the adornment in the former.* □

Algorithm 6.3.1 *SUBSUME*
Input: Two templates t and t'
Output: 1 if t subsumes t'; else 0
Method:
For each attribute A in t and t'
  Let x be the adornment of A in t'
  Let y be the adornment of A in t
  If (Table 6.1$[x, y]$ is not yes)
    Return(0)
Return(1) $\diamond$

Figure 6.1: Algorithm *SUBSUME*.

Proof: Consider two templates t and t'. If for each attribute the adornment in t' is at least as restrictive as the adornment in t, it is clear that every query supported by t' is also supported by t. If there is an attribute (say A) for which the adornment in t' is not at least as restrictive as the adornment in t, we can construct a query that is supported by t' and not supported by t. The query specifies the A attribute in such a way that the specification satisfies the restriction of the adornment of A in t' and does not satisfy the restriction of the adornment of A in t.

Thus, t subsumes t' exactly when each attribute is at least as restrictive in t' as it is in t. ∎

Figure 6.1 shows an efficient algorithm called SUBSUME for testing if a given template t subsumes another given template t'. SUBSUME simply checks if for each attribute, t' is at least as restrictive as t, by looking up Table 6.1. The number of lookups to this table is k, where k is the number of attributes in the templates. Each lookup may take $O(m)$ time where m is the maximum size of a menu in the two templates.[1] Thus, we see that algorithm SUBSUME ascertains if a template subsumes another template in $O(km)$ time.

6.3.1 Algorithm SIMPLE

Based on the notion of template subsumption, we devise an algorithm called SIMPLE, for computing the most concise design equivalent to a given design. As shown in Figure 6.2, given a set of templates, algorithm SIMPLE identifies and eliminates templates that are subsumed by other templates in the set.

[1]We assume for simplicity that operations like checking if a menu is a subset of another menu take time that is proportional to the size of the menus.

Algorithm 6.3.2 *SIMPLE*
Input: A set of templates T
Output: A minimized set of templates S
Method:

$S \leftarrow T$
For each $t \in S$
  For each $t' \in T$
    If (SUBSUME(t', t))
      $S \leftarrow S - \{t\}$
Return(S) $\diamond$

Figure 6.2: Algorithm *SIMPLE*.

Algorithm SIMPLE conducts $O(n^2)$ checks of template subsumption, where n is the number of templates in its input set. It then filter out the subsumed templates in $O(n)$ time. Each check for template subsumption takes $O(km)$ time. Thus, SIMPLE runs in $O(n^2km)$time.

We illustrate algorithm SIMPLE by way of an example.

EXAMPLE 6.3.1 Given the set of templates $\{bfu, ubf, bbu\}$, algorithm SIMPLE identifies that the bbu template is subsumed by the bfu template. Consequently, it eliminates the bbu template and arrives at $\{bfu, ubf\}$, as neither of the remaining templates is subsumed by any template in the given set.

Now, consider the set of templates $\{bfu, ubf, fbu\}$. As before, the first two templates are not subsumed by any template in the set. The third template fbu is also not subsumed by either of the other two templates. So, algorithm SIMPLE returns the original set of three templates. However, we can see that every query supported by the fbu template is also supported by one of bfu or ubf templates. That is, the fbu template is actually redundant with respect to the other templates. So, there is a set of two templates $\{bfu, ubf\}$ that is equivalent to the original set. $\square$

Although algorithm SIMPLE is efficient, it has the drawback of not being able to obtain optimal solutions to the template-set minimization problem (TSMP) in all cases, as illustrated by the above example. However, there are well-defined classes of templates for which SIMPLE is guaranteed to produce optimal solutions. Moreover, some of these classes of templates are very useful in the sense that they occur in practical systems dealing with query interfaces of Web sources.

The first class of restricted templates we consider is the class of *atomic* templates. Templates in this class are restricted to use only the b, u and c adornments. Moreover, the c adornments can only have single constants in their menus. There are two reasons why we study the class of atomic templates. First, atomic templates are fundamental – a combination of atomic templates can express any template over the set of all adornments; an atomic template cannot be expressed as a combination of any other templates. Second, atomic templates are used in many practical systems, like TSIMMIS [29].[2]

We first identify an interesting property of atomic templates and then use this property to assert that algorithm SIMPLE yields *optimal* solutions to TSMP for the class of atomic templates.

Lemma 6.3.2 *Given an atomic template t and a set of atomic templates T, the set of supported queries of t is a subset of the set of supported queries of T if and only if t is subsumed by some template in T.* □

Proof: It is obvious that if t is subsumed by some template in T, the set of supported queries of t is a subset of the set of supported queries of T.

If t is not subsumed by any template in T, we construct a query Q that is supported by t and not by T. Query Q specifies a unique value (which does not appear in any of the constant menus of the templates in T) for all the attributes that have the b adornment in t. Attributes that have the u adornment in t are left unspecified by Q. In the case of attributes with c adornments in t, Q specifies the required constant values. Obviously, Q is supported by t.

If a template in T were to support Q, it must have the u adornment for attributes left unspecified by Q, b adornment for the attributes that have the unique value specified by Q and either a b adornment or a c adornment (with matching constants) for other attributes (that had the c adornments in t). Clearly, such a template subsumes t. Since T must have a template that supports Q in order for Q to be in the set of supported queries of T, we conclude that the set of supported queries of T is not a superset of the set of supported queries of t if T contains no template that subsumes t. ■

[2] In TSIMMIS, access to data sources is through *wrappers* that translate user queries to those supported by the underlying sources. Wrappers can sometimes perform special postprocessing steps on the results of source queries and hence convert the atomic templates of the source query interfaces into more powerful templates.

Theorem 6.3.1 *Algorithm SIMPLE yields optimal solutions to TSMP for the class of atomic templates.* □

Proof: The set of templates obtained by SIMPLE cannot support more queries than the original set because the former is a subset of the latter. It is also easy to see that the set obtained by SIMPLE supports all the queries that the original set supports because every template that is dropped by SIMPLE must have a subsuming template in the remaining set obtained by SIMPLE (note that template subsumption is a transitive property). Thus, SIMPLE returns a set of templates that is equivalent to the original set.

Next, for the class of atomic templates, we show that the set returned by SIMPLE is the smallest set equivalent to the original set. Given a set of templates T, let S be the set of templates obtained by SIMPLE and let M be a minimum set of templates equivalent to T. Consider a template t in S that is not in M. We know that every query supported by t must also be supported by M (because S and M are equivalent). Therefore, based on Lemma 6.3.2, there must exist a t' in M that subsumes t. Note that t' must be different from t because t is not in M. Since S is equivalent to M, every query supported by t' is also supported by S. Therefore, once again based on Lemma 6.3.2, there must exist a t'' in S that subsumes t'. Note that t'' has to be different from t because t is different from t' and t' subsumes t. Since subsumption is transitive, we conclude that there is a template t'' in S that subsumes another template t in S. This conclusion is impossible because S cannot contain templates that are subsumed by others in S. Thus, we contradict our original assumption that there is a template t that is in S and that is not in M. In other words, M must be a superset of S and so the set of templates obtained by algorithm SIMPLE is indeed the smallest set equivalent to the original set of templates. ∎

In systems like TSIMMIS, *wrappers* are deployed on top of data sources with limited query capabilities. An important function of the wrappers is to enhance the query capabilities of the underlying data sources by providing additional filtering steps. For instance, if a source does not allow the specification of an attribute value in a query (i.e., it has the u adornment), the wrapper may enhance the capability by allowing the attribute value to be specified. When a user submits such a query, the wrapper can send a corresponding query to the data source by not specifying values for attributes with the u adornment and then postprocess the results of the source query by applying filtering conditions with respect to the values for these attributes specified in the user query. Essentially, the set of templates exported by the wrapper are just those that it obtains from the data sources, with the u adornments replaced by the f adornments. That is, wrappers with postprocessing capabilities use templates with b, f and c adornments (with singleton menus) when the corresponding sources use atomic templates. We call this new class of templates the *filter* templates. Like in the case of the atomic templates, algorithm SIMPLE is guaranteed to produce optimal results for filter templates.

Theorem 6.3.2 *Algorithm SIMPLE yields optimal solutions to TSMP for the class of filter templates.* □

Proof: The proof is similar to that of Theorem 6.3.1 which established the ability of SIMPLE to yield optimal solutions to TSMP for atomic templates. ∎

The above result is important for two reasons. First, we noted that systems employing wrappers to postprocess the results of source queries use filter templates to express the enhanced query interfaces. For such systems, we can use SIMPLE to efficiently design concise query interfaces. Second, many systems and query-processing frameworks – traditional [14, 51] as well contemporary [17, 80] – use templates from a subclass of filter templates. In particular, they use only b and f adornments. In such cases, algorithm SIMPLE can efficiently obtain concise interfaces.

There are other restricted classes of templates for which we can show that algorithm SIMPLE guarantees optimal solutions to TSMP. However, for the general class of unrestricted templates, we have seen that algorithm SIMPLE can yield suboptimal solutions to TSMP.

6.4 Irredundant Template Sets

In the previous section, we have seen an efficient algorithm called SIMPLE that guarantees optimal designs if the templates involved are of certain kinds. However, in the general case, there is no nontrivial bound on how bad the solution yielded by SIMPLE can be, when compared with the optimal solution. In this section, we develop an algorithm called SPLITCHECK that guarantees certain quality of output. SPLITCHECK achieves this guarantee, based on a stronger notion of template *redundancy* defined as follows.

We say that a template t is redundant in a set of templates T if every query supported by t is also supported by at least one other template in T. The notion of template redundancy is similar to but subtly different from the notion of template subsumption. Specifically, if a template t is subsumed by another template t' in a set of templates T, it is clear that t

is redundant in T. However, t can be redundant in T even if T does not have any other template t' that subsumes t.

Algorithm SIMPLE attempted to solve the problem of minimizing a given template set by identifying and eliminating templates in the set that are subsumed by other templates in the set. Consequently, SIMPLE may end up with a template set that has redundant templates in it. Algorithm SPLITCHECK goes beyond SIMPLE to identify and eliminate templates that are redundant in the given set, thus guaranteeing that the yield is an irredundant set of templates. In a sense, we can think of SIMPLE also trying to eliminate redundant templates, but using the simpler notion of subsumption to approximate redundancy. Consequently, SIMPLE is an efficient approximation algorithm to obtain irredundant sets of templates.

6.4.1 Detecting Template Redundancy

Formally, we define the Template Redundancy Problem (TRP) as follows: given a template t and a set of templates T, is t redundant with respect to T? As indicated by the following theorem, identifying template redundancy (i.e., solving TRP) is much harder than checking for template subsumption.

Theorem 6.4.1 *TRP is $\mathcal{NP}$-hard.* □

Proof: The proof is by reducing CNF-SAT to TRP. Given a CNF expression, we create a template corresponding to each template in a way that is similar to the proof of Theorem 6.2.1 – b adornment for positive literal, u adornment for negative literal and f adornment if the variable is absent in the clause. The set of clauses in the CNF expression yield the set of templates T. We create another template t that has the f adornment for all attributes.

Using an argument similar to that in Theorem 6.2.1, we can show that the CNF expression is satisfiable if and only if t is not redundant with respect to T. ∎

Even though in general it is very hard to test for redundancy of templates, for specific classes of templates the problem is tractable. In particular, we have the following variation of Lemma 6.3.2 that allows us to efficiently test for template redundancy in the case of atomic templates.

Lemma 6.4.1 *An atomic template is redundant with respect to a set of (atomic and non-atomic) templates if and only if it is subsumed by at least one of those templates.* □

Algorithm 6.4.1 *ATOMIZE*
Input: A template t
Output: A set of atomic templates equivalent to t
Method:

$D \leftarrow \{t\}$
While (D changes)
  $t' \leftarrow$ the first nonatomic template of D
  $D \leftarrow D - \{t'\}$
  If (t' has an f adornment)
    Let A be an attribute with f adornment in t'
    Create d_1 as a copy of t' except it has b adornment for A
    Create d_2 as a copy of t' except it has u adornment for A
    $D \leftarrow D \cup \{d_1\}$
    $D \leftarrow D \cup \{d_2\}$
  Else If (t' has an o adornment)
    Let A be an attribute with o adornment in t'
    Create d_1 as a copy of t' except it has u adornment for A
    Create d_2 as a copy of t' except it has c adornment with same menu of t' for A
    $D \leftarrow D \cup \{d_1\}$
    $D \leftarrow D \cup \{d_2\}$
  Else If (t' has a c adornment with multiple-choice menu)
    Let A be an attribute with a c adornment having a multiple-choice menu M in t'
    For each choice $c_i \in M$
      Create d_i as a copy of t' except with menu of $\{c_i\}$ instead of M for A
      $D \leftarrow D \cup \{d_i\}$
Return(D) $\diamond$

Figure 6.3: Algorithm *ATOMIZE*.

Proof: The proof is similar to that of Lemma 6.3.2. ∎

It turns out that every non-atomic template can be split into an equivalent set of atomic templates (see Lemma 6.4.2 and Figure 6.3). Thus, we can verify the redundancy of non-atomic templates by splitting them into their constituent atomic templates and checking for their redundancy easily using Lemma 6.4.1.

Lemma 6.4.2 *Every template can be split into a set of atomic templates such that the resulting set of atomic templates supports the same set of queries as the original template.*
□

Algorithm 6.4.2 *REDUNDANT*
Input: A template t and a set of templates T
Output: 1 if t is redundant in T, 0 otherwise
Method:

```
D_t ← ATOMIZE(t)
For each d ∈ D_t
    For each t' ∈ T
        If (SUBSUME(t', d))
            D_t ← D_t − {d}
If (D_t is empty)
    Return(1)
Else
    Return(0) ◇
```

Figure 6.4: Algorithm *REDUNDANT*.

Proof: We prove this lemma constructively. That is, we construct a set of atomic templates starting with a given template and argue that the resulting set has the same supported queries as the original template. Whenever the given template has an f adornment for an attribute, we split that into two templates, one with a b adornment and the other with a u adornment for that attribute. The split templates copy the adornments of all the other attributes from the original template. It is obvious that such a replacement of the original template with the two constituent templates preserves the set of supported queries. If the template has an o adornment, we split it into two templates with the u and the c adornments. The menu of the o adornment in the original template is copied over to the c adornment in the split template. Once again, the original template is equivalent to the combination of the two templates it is split into. Continuing in this manner, we end up with a set of templates that do not have any f and o adornments. Then, we replace each template in this set that has a c adornment for an attribute with a set of templates that have c adornments with singleton menus for that attribute. Thus, we construct a set of atomic templates that are equivalent to a given template. ∎

Based on Lemma 6.4.1 and Lemma 6.4.2, we construct algorithm REDUNDANT (see Figure 6.4) to test for the redundancy of a template t with respect to a set of templates T. Algorithm REDUNDANT uses algorithm ATOMIZE (see Figure 6.3) to enumerate the set of constituent atomic templates of t and checks their subsumption by the templates in T.

Algorithm 6.4.3 *SPLITCHECK*
Input: A set of templates T
Output: An irredundant subset S of T
Method:
$S \leftarrow T$
For each $t \in S$
If (REDUNDANT$(t, S - \{t\})$)
$S \leftarrow S - \{t\}$
Return(S) $\diamond$

Figure 6.5: Algorithm *SPLITCHECK*.

If all the atomic constituents of t are subsumed by templates in T, REDUNDANT declares that t is redundant with respect to T.

The worst-case running time of REDUNDANT is $O((m+2)^k nkm)$ where m is the maximum size of any menu in t, n is the number of templates in T and k is the number of attributes over which the templates are defined. The complexity expression is derived by noting that ATOMIZE finds $O((m+2)^k)$ constituent atomic templates for t and for each of these constituents REDUNDANT tests for subsumption with respect to some template in T (which has n templates and each subsumption test takes $O(km)$ time).

6.4.2 Eliminating Redundant Templates

Given a set of templates, algorithm SPLITCHECK (see Figure 6.5) uses algorithm REDUNDANT to identify and eliminate redundant templates.

EXAMPLE 6.4.1 As in Example 6.3.1, consider the set of templates $\{bfu, ubf, fbu\}$. Recall that algorithm SIMPLE could not identify that fbu is redundant in this set of templates because it is not subsumed by either of the other two templates. That is, SIMPLE returns the given set as its result.

Algorithm SPLITCHECK examines each template for redundancy. Template bfu is not redundant because one of its atomic constituents, buu, is not subsumed by either ubf or fbu. Similarly, ubf is not redundant because one of its atomic constituents ubb is not subsumed by either bfu or fbu. However, after splitting the third template fbu into $\{bbu, ubu\}$, we can ascertain that both its atomic constituents are subsumed (bbu by bfu and ubu by

ubf). Therefore, SPLITCHECK correctly identifies that fbu is redundant and eliminates it. In the next iteration, neither of the remaining two templates is redundant and hence SPLITCHECK returns the irredundant subset $\{bfu, ubf\}$. □

Given a set of templates T, algorithm SPLITCHECK runs in time that is polynomial in n, the cardinality of T, and exponential in k, the number of attributes in t. The first step of mimicking SIMPLE to eliminate subsumed templates takes $O(n^2km)$ time. The number of iterations to check and eliminate redundant templates is bounded by n. Each iteration is dominated by the call to REDUNDANT which runs in $O((m+2)^k nkm)$ time. Thus, the worst-case running time of SPLITCHECK is $O(n^2km + n(m+2)^k nkm)$.

Even though SPLITCHECK has a worst-case time complexity that is exponential in its input size, it can be very useful in practice. Eliminating redundant templates is done in an "offline" manner, at the time of design (not at query-execution time). Often it is the case that the benefits of a good design far outweigh the potentially long time SPLITCHECK takes to run.

6.5 Template Difference

The exponent in the running-time complexity of algorithm SPLITCHECK is the number of attributes, not the number of templates or the number of choices in menus. Thus, SPLITCHECK performs very well when the number of attributes is moderately small. However, its performance can be poor in cases where the number of attributes is large, even though the number of templates may remain small. In this section, we present another algorithm, named DIFFCHECK, that produces irredundant template sets in time that is polynomial in the number of attributes.

As in the case of SPLITCHECK, algorithm DIFFCHECK attempts to minimize the template set by finding and eliminating redundant templates. However, it uses a new approach to identify the redundant templates. At the heart of this approach lies the notion of computing the difference of one template with respect to another.

Essentially, given two templates t_1 and t_2, $t_1 - t_2$ allows precisely the set of queries supported by t_1 and not supported by t_2. The difference, $t_1 - t_2$, is expressed as a set of templates that supports these queries. The following example illustrates the process of computing template difference.

EXAMPLE 6.5.1 Consider two templates bfu and fbu. Obviously, a query that specifies a value for the first attribute and leaves unspecified the other two attributes is supported by bfu and not by fbu. Such a query is allowed by $bfu - fbu$. In contrast, a query that specifies values for the first two attributes and leaves unspecified the third should not be in $bfu - fbu$. Such a query is allowed by the first template bfu, but it is also allowed by the second template fbu.

It turns out that the difference of bfu with respect to fbu is $\{buu\}$. In this case the difference is a singleton set. In other example scenarios, the difference of two templates can be a set of templates. For instance, the difference of bff with respect to fbu is $\{buf, bfb\}$. □

When templates have adornments with menus of constants, the difference computation is complicated. For instance, how do we represent the difference of bub with respect to $c[S]ub$? The result must require that queries specify a value for the first attribute, but the value cannot be in S. In the notation for attribute adornments presented in Chapter 5, there is no way to specify such a requirement. In order to facilitate such specification we adopt the notation that $c[S']$ represents the restriction that the attribute must be specified with a value that is not in S'. Similarly, $o[S']$ allows the attribute to be left unspecified or to have a value outside of S'. With the new notation, we can express the difference of template bub with respect to the template $c[S]ub$ as the template $c[S']ub$.

We present an efficient algorithm, named DIFFERENCE, to compute the difference of a template with respect to another. As shown in Figure 6.6, the algorithm proceeds one attribute at a time. It uses Table 6.2, to compute the difference between the adornments of an attribute in the two templates. If this difference is not $-$, the NULL adornment, algorithm DIFFERENCE generates a new template by copying over the adornments of all other attributes from the first template and specifying the computed difference adornment for the attribute under consideration. In general, the difference of one template with respect to another can contain up to k templates where k is the number of attributes. The difference may contain 0 templates if the first template is subsumed by the second. In this case, all calls to Table 6.2 by algorithm DIFFERENCE will result in $-$, the NULL adornment.

Based on the computation of the difference of two templates, algorithm DIFFCHECK (shown in Figure 6.7) identifies and eliminates redundant templates. Specifically, it considers each template in a given set for its redundancy test. It ascertains that a target template is redundant with respect to a reference set of templates by progressively computing the

Algorithm 6.5.1 *DIFFERENCE*
Input: Two templates t and t'
Output: The set of templates representing the difference of t with respect to t'
Method:

$D \leftarrow \{\}$
For each attribute A
    If (DiffAdorn[$adorn(A,t), adorn(A,t')$] is not $-$)
        Construct a new template t''
        $adorn(A,t'') \leftarrow$ DiffAdorn[$adorn(A,t), adorn(A,t')$]
        For each attribute A' that is not A
            $adorn(A',t'') \leftarrow adorn(A',t)$
        $D \leftarrow D \cup \{t''\}$
Return(D) $\diamond$

Figure 6.6: Algorithm *DIFFERENCE*.

	f	$\mathbf{o}[S_{21}]$	**b**	$\mathbf{c}[S_{22}]$	**u**
f	-	$\mathbf{c}[S'_{21}]$	**u**	$\mathbf{o}[S'_{22}]$	**b**
$\mathbf{o}[S_{11}]$	-	$\mathbf{c}[S_{11} - S_{21}]$	**u**	$\mathbf{o}[S_{11} - S_{22}]$	$\mathbf{c}[S_{11}]$
b	-	$\mathbf{c}[S'_{21}]$	-	$\mathbf{c}[S'_{22}]$	**b**
$\mathbf{c}[S_{12}]$	-	$\mathbf{c}[S_{12} - S_{21}]$	-	$\mathbf{c}[S_{12} - S_{22}]$	$\mathbf{c}[S_{12}]$
u	-	-	**u**	**u**	-

Table 6.2: Computing Difference of Two Adornments: DiffAdorn.

Algorithm 6.5.2 *DIFFCHECK*
Input: A set of templates T
Output: An irredundant subset S of T
Method:

```
S ← T
For each t ∈ S
    D ← {t}
    For each t′ ∈ S, such that t′ is not t
        For each t″ ∈ D
            If (t″ not in DIFFERENCE(t″, t′))
                D ← D − {t″}
                D ← D ∪ DIFFERENCE(t″, t′)
    If (D is empty)
        S ← S − {t}
Return(S) ◇
```

Figure 6.7: Algorithm *DIFFCHECK*.

residue of the target template with respect to each of the reference templates. It starts with the target template as the original residue (a set of templates). Then, it replaces this template by its difference with respect to the first reference template. Subsequently, it replaces each of the residual templates by their difference with respect to the other reference templates. At any point of time, the residual set of templates represents the set of queries supported by the target template and not supported by any of the reference templates considered so far. If the residue is empty (i.e., there are no residual templates), after all the reference templates have been considered, algorithm DIFFCHECK concludes that the target template is redundant and eliminates it.

Algorithm DIFFCHECK runs in time that is polynomial in the number of attributes k and exponential in the number of templates n. Specifically, it has $O(n)$ iterations of testing for redundancy of target templates. Each iteration computes the residual templates over $O(n)$ layers. Each layer can expand the residual template set k-fold. Thus, we arrive at a complexity of $O(k^n)$ templates in the residual set. Hence, the running time is polynomial in k and exponential in n.

6.6 Template Coalescing

Algorithms SPLITCHECK and DIFFCHECK yield irredundant designs. Nevertheless, there may still be opportunities to find more concise designs. For instance, consider the set: $\{bbuf, bbbf, buff, bfbb\}$. The only template that is redundant in this set is $bfbb$ and so we can arrive at the irredundant design: $\{bbuf, bbbf, buff\}$. However, a more concise equivalent design is: $\{bfff\}$. This design can be obtained by recognizing that we can *coalesce* the templates of the earlier design, two at a time. For example, $bbuf$ and $bbbf$ coalesce to form $bbff$, which in turn coalesces with $buff$ to form $bfff$. In a sense, the process of coalescing templates is inverse to the process of splitting templates employed by algorithm SPLITCHECK (to identify the redundancy of "composite" templates).

If it were true that we can always coalesce two templates to form a single equivalent template, we can minimize any given set of templates into a singleton set. However, it is not always possible to coalesce a pair of templates, as illustrated by the following example.

EXAMPLE 6.6.1 Consider two templates: bub and ubb. These two templates cannot be coalesced into a single template. That is, there does not exist any template that is equivalent to the combination of these two templates. Note that it is quite easy to find a template that subsumes the two given templates, but such a template will allow queries not allowed by either of the two given templates. For instance, the template ffb subsumes both the templates, but it allows a query that specifies only the third attribute. This query is not allowed by either of the two given templates we started with. □

How can we determine if two templates can be coalesced? Fortunately, as with template subsumption, we can ascertain efficiently the ability to coalesce a given pair of templates. The following lemmas specify precisely the conditions for a pair of templates to be coalesced.

Lemma 6.6.1 *Two templates can be coalesced if they differ in at most one attribute adornment.* □

Proof: If two templates do not differ in any attribute adornment, then they are indeed the same templates. These two templates can trivially be coalesced into a single template, which has the same attribute adornments as the given templates. Now, consider two different templates t and t' that differ in the adornment of exactly one attribute. We show that t and t' can be coalesced into a single template s as follows. Let A be the attribute in whose adornment t and t' differ. We construct s

by copying over the adornment of t for all attributes other than A. For A, the adornment in s is the combination of the adornments in t and t' according to Table 6.3. By examining the entries in Table 6.3, one can see that the adornment of A in the combined template is not more restrictive than the adornments of A in the templates being coalesced. For instance, when t has the b adornment and t' has the u adornment, s ends up with the f adornment, one that is less restrictive than both the b and the u adornments. Since s will have an adornment for each attribute that is not more restrictive than either of the two adornments in t and t' for that attribute, we conclude that s subsumes both t and t'. It remains to be shown that s does not allow any query that is not allowed by t and t'. Once again, a close examination of the entries in Table 6.3 reveals that there cannot be a query which satisfies the adornment of A in s, while at the same time does not satisfy either of the adornments of A in t and t'. For instance, the $o[S_{12}]$ entry for $c[S_{12}]$ and u indicates that every query that satisfies the $o[S_{12}]$ adornment of s either satisfies the $c[S_{12}]$ adornment of t or the u adornment of t'. Since s subsumes both t and t' and does not allow any query that is not allowed by either t or t', s is equivalent to the combination of t and t'. Thus, t and t' are coalesced to form s. ∎

Lemma 6.6.2 *Two templates that differ in more than one attribute can be coalesced if and only if one subsumes the other.* □

Proof: It is obvious that if a template subsumes another, we can trivially coalesce the two templates to form a copy of the subsuming template. We now show that subsumption is a necessary condition for coalescing two templates that differ in more than one attribute adornment. Consider two templates t and t' that differ in more than one attribute adornment, such that neither t nor t' subsumes the other. Let s be the template obtained by coalescing t and t'. We will show that s cannot exist.

We consider two cases for the attribute-adornment difference in t and t'. The first case is when there is an attribute A for which t has an adornment that is more restrictive than the adornment in t' or vice versa. The second case is when no such attribute exists. In the first case, without loss of generality, let the adornment of A in t be more restrictive. Then, the adornment of A in s must be the same as the adornment in t' (in order for s to cover t' it cannot be more restrictive, and in order for s to be equivalent to the combination of t and t' it cannot be less restrictive). Since t and t' differ by more than one attribute and since t' does not subsume t, there must be another attribute B for which t has an adornment that is less restrictive than the adornment in t'. In other words, there must be a query whose specification for B is such that the query is allowed by t and not by t'. Since s subsumes t, this query must be allowed by s. Now, we can modify this query to specify A in such a way that t cannot allow the modified query and yet s allows the modified query (because s has an adornment for A that is strictly less restrictive than t). Because of the specification of the

Algorithm 6.6.1 *COALESCE*
Input: Two templates t and t'
Output: 1 if t and t' can be coalesced, 0 otherwise
Method:
Let t and t' differ in X attribute adornments
If $((X \leq 1)$ or SUBSUME(t, t') or SUBSUME$(t', t))$
Return(1)
Else
Return(0) $\diamond$

Figure 6.8: Algorithm *COALESCE.*

B attribute, the modified query is disallowed by t' too. Thus, we see that there is a query allowed by s that is not allowed by t and t'.

In the second case, there is no attribute for which t is more restrictive than t' or vice versa. Let A and B be two attributes on which t and t' differ. We can construct a query that specifies A in such a way that t allows the query and t' disallows the query. Since s must subsume t, this query has to be allowed by s. We can then modify this query by specifying B so that t disallows the modified query. The modification to the specification of B can be done in such a way that the new specification satisfies the adornment of B in t' and consequently s. In this way, we again construct a query that is allowed by s and not allowed by both t and t'.

Thus, we have shown that there cannot be an s that subsumes both t and t' while not allowing any query disallowed by both t and t'. Hence, two templates that differ in more than one attribute adornment can only be coalesced if one subsumes the other. $\blacksquare$

Based on the above two lemmas, we develop an efficient algorithm to determine the possibility of coalescing a pair of templates. Algorithm COALESCE (presented in Figure 6.8) simply checks for the two positive cases: a) the templates differ in at most one attribute adornment, b) one template subsumes the other. If either of the two cases holds, COALESCE declares that the given templates can be coalesced, otherwise it correctly determines that the templates cannot be coalesced.

Algorithm COALESCE runs in $O(km)$ time, where k is the number of attributes and m is the size of the largest menu of choices in the given pair of templates.

Once we determine that a pair of templates can be coalesced, the actual process of coalescing the two templates is fairly simple. The resultant template constructs the adornment

	f	$\mathbf{o}[S_{21}]$	**b**	$\mathbf{c}[S_{22}]$	**u**
f	**f**	**f**	**f**	**f**	**f**
$\mathbf{o}[S_{11}]$	**f**	$\mathbf{o}[S_{11} \cup S_{21}]$	**f**	$\mathbf{o}[S_{11} \cup S_{22}]$	$\mathbf{o}[S_{11}]$
b	**f**	**f**	**b**	**b**	**f**
$\mathbf{c}[S_{12}]$	**f**	$\mathbf{o}[S_{12} \cup S_{21}]$	**b**	$\mathbf{c}[S_{12} \cup S_{22}]$	$\mathbf{o}[S_{12}]$
u	**f**	$\mathbf{o}[S_{21}]$	**f**	$\mathbf{o}[S_{22}]$	**u**

Table 6.3: Combining Adornments from Coalesced Templates.

Algorithm 6.6.2 *CONDENSE*
Input: A set of templates T
Output: An equivalent set of templates S
Method:
$S \leftarrow T$
While (COALESCE(t, t') for some $t, t' \in S$)
$S \leftarrow S - \{t\}$
$S \leftarrow S - \{t'\}$
Add to S the result of coalescing t and t'
Return(S) $\diamond$

Figure 6.9: Algorithm *CONDENSE*.

of each attribute by consulting Table 6.3 with the adornments of the attribute in the given two templates.

Now that we know how to combine two templates, we can devise an algorithm that can incrementally condense a set of templates. Algorithm CONDENSE, shown in Figure 6.9, considers pairs of templates to be coalesced in an iterative manner. That is, in each iteration, CONDENSE finds a pair of templates that can be coalesced and replaces the two templates by the template that results from coalescing the two templates. When no pair of templates can be coalesced, CONDENSE returns the remaining set of templates.

EXAMPLE 6.6.2 Consider the set of templates: $\{bbb, bub, ubb, uub\}$. Algorithm CONDENSE can first identify that the bbb and bub templates can be coalesced and then shrink the set to $\{bfb, ubb, uub\}$. In the next iteration, it can coalesce ubb and uub to arrive at $\{bfb, ufb\}$. Finally, it can then coalesce these templates to arrive at the concise singleton set: $\{ffb\}$.

Consider another set of templates: $\{ubb, bbb, bub, ufb\}$. Let CONDENSE first consider the pair ubb and bbb and coalesce the two to obtain $\{fbb, bub, ufb\}$. At this stage, every pair of templates differs by at least 2 attribute adornments and no template subsumes any other. So, we cannot coalesce any more pairs of templates and CONDENSE returns the design of three templates.

However, the given set $\{ubb, bbb, bub, ufb\}$ is again equivalent to the singleton set: $\{ffb\}$. First, note that the ffb template subsumes each of the four templates in the starting set. So, it is at least as powerful as the given set of templates. Now, we need to determine if $\{ffb\}$ is too powerful? It can be split into its equivalent set of atomic templates: $\{bub, ubb, bbb, uub\}$. We can see that the first three templates here are also in the starting set and the fourth one is covered by the fourth template in the starting set. Therefore, $\{ffb\}$ is not more powerful than the starting set. Hence, it is equivalent to the starting set of templates. Thus, we have a concise singleton set equivalent to the starting set, while CONDENSE ended up with a design of three templates.

Starting with $\{ubb, bbb, bub, ufb\}$, if CONDENSE first coalesced bub and bbb, then coalesced ubb and ufb, and then coalesced the resulting two powerful templates bfb and ufb, it would have arrived at the singleton set $\{ffb\}$. In summary, CONDENSE can sometimes choose the "wrong" pairs of templates to coalesce and end up with suboptimal solutions. □

As illustrated by Example 6.6.2, algorithm CONDENSE is not guaranteed to find the smallest equivalent set. This fact is not surprising since CONDENSE runs in $O(n^3km)$ time while the problem it is trying to solve, namely TSMP, is intractable.

6.7 Related Work

Query templates using attribute adornments have been employed to describe sets of answerable queries in many research prototypes [14, 28, 29, 40, 51, 72], and recent literature [17, 59, 80]. The main emphasis of this body of research is on issues surrounding query planning, as opposed to interface design. By focusing on the problem of efficient interface design for templates, the solutions developed in this chapter complement this body of research. Our solutions can be employed by systems like TSIMMIS [29] and GARLIC [28] to design powerful interfaces to their data sources and use the concise template sets so obtained to efficiently plan the query execution over these sources. For instance, in Section 6.3 we developed a very efficient algorithm that yields the most-concise template sets for data sources in TSIMMIS [29].

The sets of interface adornments used in most previous systems are very limited. For instance, [17] and [80] use only b and f adornments. Data sources on the Web have a wide range of query-processing capabilities and interface requirements from their target customers. This scenario is evidenced by the large number and variety of search forms on the Web (e.g., [85, 90, 97]). To describe this variety of query interfaces, extended sets of attribute adornments, and templates based on these adornments, are proposed in recent literature [82]. In studying the problem of obtaining concise query interfaces, we adopt the extended set of adornments of [82] instead of the simpler sets of adornments of frameworks like [17] and [80].

Mediation systems extend the limited capabilities of data sources they work with [12, 14, 28, 29]. Essentially, the templates of mediators will be more flexible than the corresponding data sources, depending on the abilities of such systems to extend the source capabilities. Recent work [82] has shown how to compute the capabilities of mediators, based on the capabilities of their sources. During the computation of the set of templates for a mediator, we may end up with large template sets when the mediator is integrating information over many sources. Large template sets lead to undesirable consequences like inefficient query processing and cumbersome user interfaces. The techniques we developed here are very useful in alleviating the problems with complex interfaces of large-scale mediators.

Some of the results discussed in this chapter had been published in a Stanford University Technical Report [83].

6.8 Chapter Summary

In this chapter, we presented a formal framework to describe the query capabilities of data sources. Based on this framework, we studied the problem of obtaining concise query interfaces for data sources. In particular, we formally defined the problem of minimizing template sets (TSMP); showed that TSMP is intractable; developed an efficient approximation algorithm, called SIMPLE, that guarantees optimal solutions in many practical instances of TSMP; presented other algorithms, namely SPLITCHECK and DIFFCHECK, that guarantee the generation of irredundant template sets in all cases; discussed a technique of combining templates that can be used to postprocess the output of SPLITCHECK and DIFFCHECK to obtain more concise template sets.

Chapter 7

The TSIMMIS System

7.1 Overview

At Stanford University, we built a research prototype named TSIMMIS that incorporates many ideas from our study of problems and solutions in the areas of data integration and query processing. The TSIMMIS system integrates data from multiple heterogeneous sources and provides users with seamless integrated views of the data. It uses the *mediation* architecture [79] to extend the query capabilities of data sources and to deal with limited capabilities of data sources when processing user queries on integrated views (Figure 7.1).

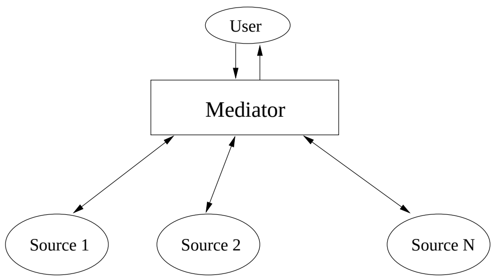

Figure 7.1: The TSIMMIS Architecture.

We note here that the development of the TSIMMIS system had been underway before research began on the issues specifically discussed in the earlier chapters of this thesis. In fact, many of the research problems addressed in this thesis had been identified during the development of the TSIMMIS system. Thus, TSIMMIS not only incorporated the solutions developed in this thesis but also helped define the problems on which this thesis is based. In this chapter, we give a brief overview of the TSIMMIS system. We emphasize the aspects of TSIMMIS that are more directly related to some of the issues discussed in this thesis (e.g., extending the capabilities of data sources through postprocessing techniques).

7.2 The TSIMMIS Mediator

The centerpiece of the TSIMMIS architecture is the mediator. In this section, we describe how the TSIMMIS mediator helps in processing user queries over a set of data sources. Specifically, we show how the mediator translates a user query into a set of relevant source queries, constructs a query plan involving the source queries, and then executes the query plan. Later on, in Section 7.3, we describe how TSIMMIS wrappers provide uniform object exchange between the mediator and the data sources.

The architecture of the TSIMMIS mediator is shown in Figure 7.2(a). The mediator consists of primarily three modules: View Expander, Plan Generator, and Execution Engine. These three modules are described in Sections 7.2.1, 7.2.2, and 7.2.7 respectively.

7.2.1 View Expander

The mediator provides integrated views across the data from the set of sources it mediates. In particular, it defines these integrated views in terms of the source views through a set of view definitions or rules. It uses the *Mediator Specification Language* (MSL) [55] to define the integrated views. MSL is a logic-based language with object-oriented features. For example, let a mediator view `paper` be defined as follows:

```
<paper {<title T> <author A> <abs B> <conf C>}> :-
       <entry {<title T> <author A> <abs B>}>@s1,
       <entry {<title T> <conf C>}>@s2
```

The above rule states that `paper` is essentially a join of the views exported by sources s_1 and s_2, with `title` being the join attribute. Each rule in MSL has a head and a tail that consist of OEM objects. OEM is a self-describing data model in which each object has a `label` field and a `value` field. The value field can be atomic or complex. Complex values accommodate object nesting. For example, in the MSL rule describing the `paper` view, the

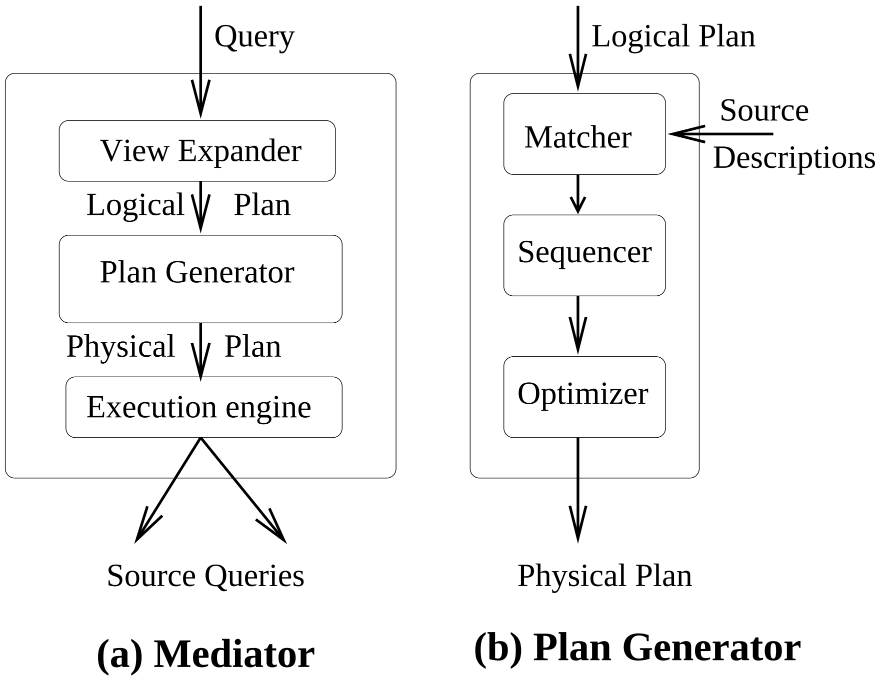

Figure 7.2: Mediator Architecture.

head consists of an OEM object with label `paper` and a list of subobjects describing the `title`, `author`, `abstract`, and `conference` of the `paper` object. Uppercase identifiers (e.g., `T, A`) are variables that are bound to specific values at run time.

Suppose we want to find the `title` and `abstract` of each paper written by "Smith" in "SIGMOD-97". We formulate the following query, based on the mediator view `paper`:

```
<ans {<title T> <abs B>}> :-
     <paper {<title T> <author ''Smith''> <abs B> <conf ''SIGMOD-97''>}>
```

When this query arrives at the mediator, the mediator uses the view definitions to translate the query on the mediator view into a *logical plan* [55] (i.e., a set of MSL rules which refer to source views instead of mediator views). This translation process is called *View Expansion* and involves the matching of the mediator views specified on the right-hand side of the user query with the left-hand side of the view definitions. When a match is found, the right-hand side of the corresponding view definition takes the place of the mediator view in the user query. The following is the logical plan for the example user query:

```
<ans {<title T> <abs B>}> :-
     <entry {<title T> <author ''Smith''> <abs B>}>@s1,
     <entry {<title T> <conf ''SIGMOD-97''>}>@s2
```

The logical query plan indicates that answers to the user query may be computed by sending two source queries. The first one, to s_1, gets the title and abstract of each entry (for a paper) corresponding to "Smith", while the second one, to s_2, gets the title of each paper in the "SIGMOD-97" conference. From the results of the two source queries, the bindings for variables T and B are obtained to construct the answers to the user query.

Note that in our example situation, the logical query plan has only one rule. The user query has only one mediator view, which has only one expansion. If the mediator view were a union view, instead of a join view, it will have multiple MSL rules defining it. In such a case, the view will have multiple possible expansions, each leading to an MSL rule in the logical query plan.

7.2.2 Plan Generator

The logical plan generated by the view expander is passed onto the Plan Generator, which computes a feasible physical plan for the query. We refer to the parts of the right-hand sides of the rules of the logical query plan as *conditions*. The logical query plans in TSIMMIS do not specify the order in which the conditions are processed. The order of processing the condition, and consequently the sequence of source queries, is specified in the physical plans constructed by the Plan Generator.

Three possible physical plans for the logical plan of the example user query are:

- P_1: Send query <entry {<title T> <author ''Smith''> <abs B>}> to s_1;[1] send query <entry {<title T> <conf ''SIGMOD-97''>}> to s_2; join the results of these source queries on the title attribute.

[1]Strictly speaking, source queries are MSL rules. For simplicity, we just show the tail of the source query rule.

- P_2: Send query <entry {<title T> <author ‘‘Smith’’> <abs B>}> to s_1; for each returned title, send query <entry {<title T> <conf ‘‘SIGMOD-97’’>}> to s_2, by binding T.
- P_3: Send <entry {<title T> <conf ‘‘SIGMOD-97’’>}> to s_2; for each returned title, send <entry {<title T> <author ‘‘Smith’’> <abs B>}> to s_1, by binding T.

Some of these physical plans may or may not be feasible depending on the query capabilities of the sources.

7.2.3 Source Capabilities

In order to describe the capabilities of sources, the TSIMMIS system uses templates to represent sets of queries that can be processed by each source [29, 57]. Suppose s_1 and s_2 have the following templates.

T_{11}: X:-X:<entry {<title $T> <author A> <abs B>}>@$s_1$

T_{21}: X:-X:<entry {<title T> <conf $C>}>@$s_2$

T_{22}: X:-X:<entry {<title $T> <conf C>}>@$s_2$

The first template T_{11} says that source s_1 can return all the information it has about a paper given its title. T_{21} says that s_2 can return all the information it has about papers given the conference. T_{22} says that s_2 can also return the information about a paper given its title. Assume that these are the only templates for s_1 and s_2. That is, s_1 and s_2 cannot answer any other kinds of queries.

Given these capabilities, P_1 (Section 7.2.2) is not feasible since s_1 cannot answer the query <entry {<title T><author ‘‘Smith’’><abs B>}>. Note that s_1 requires that the title value be specified. P_2 is also infeasible for the same reason. Only P_3 is feasible as the mediator first gets the title of each paper in "SIGMOD-97" from s_2 and uses this title value to get the corresponding abstract information from s_1 and check that the author is indeed "Smith". Notice that the queries to s_1 are now feasible because they specify the title values.

As shown in Figure 7.2(b), the Plan Generator consists of three components: Matcher, Sequencer, and Optimizer. These components help generate a physical plan for the logical plan passed into the Plan Generator.

Source Query	Source Template	Condition Processed	Binding Requirement
M_1	T_{11}	C_1	T
M_2	T_{21}	C_2	None
M_3	T_{22}	C_2	T

Table 7.1: Matcher Result

7.2.4 Matcher

The first step in the plan-generation process is to find all the source queries that can process parts of the logical plan. Some of these queries have requirements indicating the list of variables that need to be bound. To illustrate, consider the logical plan of Section 7.2.1. Let the two conditions of the logical plan be denoted C_1 and C_2. There is one way to process C_1, with the requirement that variable `T` be bound. There are two ways to process C_2: one with `T` bound and another without any binding requirements.

Table 7.1 describes the result of the Matcher. It has a row for each source query that processes some conditions of the logical plan. For instance, the first row M_1 in the table indicates that s_1 can process C_1 by using template T_{11} with the requirement that variable `T` be bound.

7.2.5 Sequencer

The second step in the plan-generation process is to piece together the source queries for processing the conditions of the logical plan in order to construct feasible plans. Here, what matters is not just the specific source queries chosen to cover all the conditions of the logical plan but also the sequence of processing these queries.

The Sequencer finds the set of feasible sequences of source queries. Each query in a feasible sequence has the property that the variables in its binding requirements are exported by the source queries that appear earlier in the sequence. For instance, in our example logical plan, the Sequencer finds that the only feasible sequence is $< M_2,\ M_1 >$, with source query M_1 being parameterized (variables bound) from the result of the source query M_2. Sequence $< M_1,\ M_2 >$, though it can also process all the conditions, is not feasible because M_1's binding requirements cannot be satisfied. Other sequences like $< M_2,\ M_3 >$ cannot process all the conditions of the logical plan.

7.2.6 Optimizer

Having found the feasible sequences of source queries, the third step of the plan-generation process is to optimize over the set of corresponding feasible plans and choose the most efficient among these. The Optimizer uses standard optimization techniques to pick the best feasible plan and translates it into a physical plan. In our example case, there is only one feasible sequence of queries $< M_2,\ M_1 >$ and this sequence leads to the physical plan P_3 of Section 7.2.2.

7.2.7 Execution Engine

As shown in Figure 7.2(b), the physical plan constructed by the Plan Generator is passed onto the Execution Engine. This module sends the source queries to the appropriate sources, collects the results of these source queries and performs the postprocessing operations. The postprocessing operations include the application of additional query conditions that need to be used to filter the results of the source queries, as well as integration operations like union and join of the source-query results. The result of each source query is a set of OEM objects and the Execution Engine computes the answer to the user query as a set of OEM objects. This set of objects is returned to the user.

7.3 The TSIMMIS Wrappers

The goal of the TSIMMIS system is to provide integrated query processing across a set of heterogeneous data sources. These sources can have different data models as well as diverse and limited query interfaces. In the TSIMMIS system, we developed wrappers that can translate between the different data models of the sources and the common data model OEM. That is, by employing wrappers around data sources, the TSIMMIS mediator can abstract itself from the different data models of the sources and instead assume that the results of any source query is simply a set of OEM objects. As part of developing the TSIMMIS system, we created wrappers that deal with sources ranging from relational databases to simple file systems. We also developed wrappers for many Web sources that export HTML data in response to queries submitted to them. The HTML data is translated by the wrappers into OEM so as to facilitate further processing of the data (e.g., applying additional filtering conditions or combining results from multiple sources).

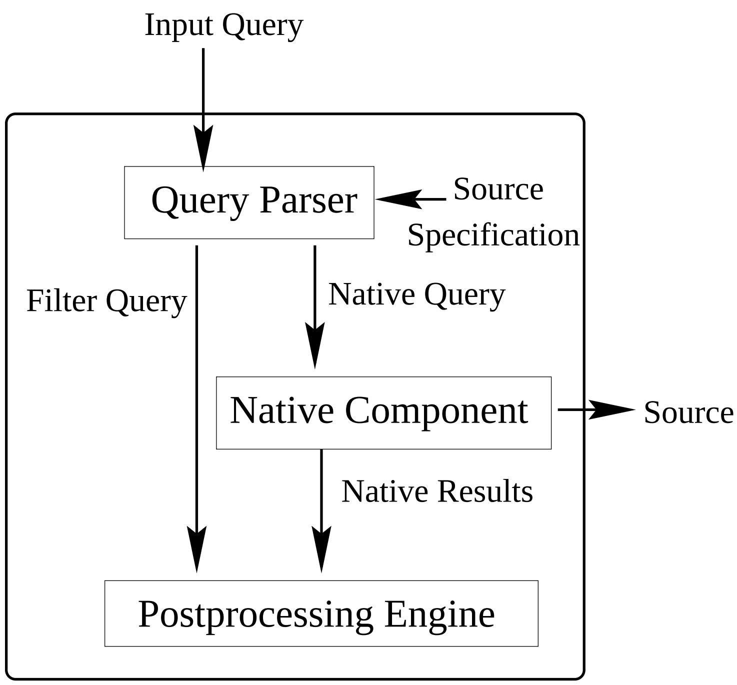

Figure 7.3: Wrapper Components.

As shown in Figure 7.3, a TSIMMIS wrapper consists of mainly three components: Query Parser, Native Component, Postprocessing Engine. These three components are described in Sections 7.3.1, 7.3.2, and 7.3.3 respectively.

7.3.1 Query Parser

In the TSIMMIS system, queries are expressed like rules in Datalog syntax. Each query tail has OEM objects with variables and constants. Conceptually, these OEM objects are matched with those at a data source and the matching source objects are returned. When matching the objects, the constants have to match exactly and the variables in the query are bound to the corresponding values in the matching objects of the data source.

The Query Parser is the first component of the TSIMMIS wrapper. It receives the query posed to the wrapper and parses it. While parsing, it consults a declarative specification of the data source to ascertain whether or not the given query is supportable. This declarative specification is in the form of a set of query templates.

Each query template has two parts. The first part describes the set of queries accepted by the data source. This specification is in the form of a rule. The right-hand side of the rule indicates the pattern of queries supported by the source. For instance, consider the following specification of a data source that stores information about conference publications.

T_1: `X:-X:<entry {<author A> <conf $C> <title T>}>`

T_2: `X:-X:<entry {<author $A> <conf C> <title T>}>`

The first template indicates that the source can find and return entries if the `conference` attribute is specified. The second template shows that the source can also support queries that specify the `author` attribute. Now, consider the following queries to the wrapper:

Q_1: `X:-X:<entry {<author A> <conf ‘‘SIGMOD-97’’> <title T>}>`

Q_2: `X:-X:<entry {<author ‘‘Smith’’> <conf C> <title T>}>`

Q_3: `X:-X:<entry {<author A> <conf C> <title ‘‘Mediation’’>}>`

The Query Parser will try to match each of the above queries with the source templates. If the query matches with a template, then it is feasible. When matching a query with a template, each constant value specified in the template must match with the corresponding constant value specified in the query and each placeholder specified in the template must have a constant value specified for the corresponding attribute in the query. For instance, Q_1 is feasible because it matches with template T_1, while Q_2 is feasible because it matches with template T_2. Query Q_3 is not feasible because it matches with none of the source templates.

Whenever a query matches with a template, a *native query* is constructed. Attached to each rule in the source specification in the TSIMMIS wrapper is a template for the corresponding source query with placeholders to be filled-in, based on the values indicated in the query. The template indicating the source query to be built can be in various forms. It can be a SQL statement if the data source is relational, it can be a URL with a CGI string if the data source has a Web form interface or it can be any other query string that the data source accepts. For instance, in the case of T_1 the source-query template can be http://source.com/process?conference=$1, where $1 will be replaced by whatever value is indicated for the `conference` attribute. Thus, the Query Parser generates a native query if the input query matches some rule in the source specification.

As opposed to queries Q_1 and Q_2 that match perfectly, consider the following query:

Q_4:
```
X:-X:<entry {<author A> <conf ``SIGMOD-97''> <title ``Mediation''>}>
```

Query Q_4 *partially* matches template T_1. That is, if it did not have the additional condition on the `title` attribute, it would have been a perfect match. Partial matches occur when the query imposes more restrictions on the answer objects than the source templates indicate. For instance, Q_4 would have matched T_1, but for the extra restriction that the answer objects must have `title` "Mediation".

The Query Parser recognizes partial matches of queries and source templates, and splits the given query into two parts, one that matches the template exactly and the other that constitutes the residual. The residual is represented in the form of a *filter query*. It is as if the wrapper first received the simpler query (without the additional conditions) that matches exactly and then on the results of this simpler query, the filter query is to be applied. In the above example, we can imagine first obtaining all entries for the "SIGMOD-97" conference and then filtering out those that are not on "Mediation". The first part is supported directly by the data source, while the second part is handled by the Postprocessing Engine (see Section 7.3.3). For the example query Q_4, the Query Parser constructs the following filter query:

F_4:
```
X:-X:<entry {<author A> <conf C> <title ``Mediation''>}>
```

7.3.2 Native Component

The Query Parser passes on the native query to the Native Component and the filter query to the Postprocessing Engine. The Native Component sends the native query to the data source. Subsequently, when the Native Component receives the results from the data source, it translates them into a set of OEM objects. In order to perform this task, the Native Component needs to know the schema of the results returned by the data source. It uses the knowledge of the schema to interpret the results (typically, data sources return their results in a character buffer) and constructs the OEM objects. It passes on the set of constructed OEM objects to the Postprocessing Engine.

7.3.3 Postprocessing Engine

The Postprocessing Engine executes the filter query on the results of the native query. Essentially, it eliminates some of the OEM objects that are output by the Native Component, as they do not satisfy the filter query.

By providing the ability to apply filtering conditions on the results of source queries, the TSIMMIS wrappers extend the query capabilities of data sources. Such postprocessing can also be undertaken at the mediator component in the TSIMMIS architecture. However, doing postprocessing at the wrapper level yields more efficient query execution as we can save on not transmitting the unnecessary OEM objects from the wrappers to the mediator.

7.4 Toolkits

In the TSIMMIS system, the mediator is built in such a way that it can be employed in a variety of contexts with different view specifications and source-capability descriptions. That is, the same code base can be deployed to create mediators in different application situations. For instance, when building a comparison-shopping application over a set of bookstore data sources, we can employ the TSIMMIS mediator code base and feed it the source-capability descriptions for the bookstore data sources and define appropriate mediator views using the source views. The capability descriptions feed into the Plan Generator module of the mediator (see Figure 7.2(a)) and the view definitions are input to the View Expander module (see Figure 7.2(a)). Thus, what we developed in the TSIMMIS project is more of a mediator toolkit that can be employed to build mediators in a variety of application contexts by providing declarative specifications to the toolkit.

In the case of wrappers, it is harder to develop a generic code base that can work with all data sources. However, the main part of the code base of the TSIMMIS wrappers (typically, more than 80%) is for the Query Parser and the Postprocessing Engine modules (see Figure 7.3), and these two modules will be the same across wrappers for different sources. The Query Parser does take as input declarative information in the form of the source templates. The Postprocessing Engine does not depend at all on the data source for which the wrapper is being constructed. The Native Component typically has to be newly written for each data source. Even in this component one can look for opportunities to reduce the need to generate large amounts of code by providing subcomponents that are common across a large number of data sources of a kind. For example, in the TSIMMIS project, we built many wrappers for data sources on the Web. All these data sources are passed in CGI strings as queries and their results are HTML pages. We built generic tools to parse HTML pages such that by merely specifying a few rules identifying the content structure of the HTML pages and the desired elements of the query results, we can create a wrapper for a Web source. Using these techniques, we could eliminate any need for code modifications when building wrappers for different data sources on the Web. That is, we can once again view the code base we built in the TSIMMIS system as a wrapper toolkit that can be used to construct wrappers for data sources on the Web by merely providing declarative information to the toolkit.

7.5 Chapter Summary

In this chapter, we presented a brief overview of the TSIMMIS system. We discussed in some detail the essential aspects of how data sources declare their query capabilities, how these capabilities are extended through postprocessing techniques and how these capabilities affect the processing of queries involving integrated views at the mediators. The development of the TSIMMIS system has helped us identify key challenges in mediation systems. For instance, the simple algorithm that the TSIMMIS system uses for capability-based mediation of join queries does not scale when large numbers of sources are involved. This problem motivated our research, the results of which are presented in Chapter 3. Similarly, the view expansion mechanism of the TSIMMIS mediators deals with fusion queries by translating them into unions of join queries. Once again, we recognized that such a strategy would not scale when moderately large numbers of sources are involved and this observation led to our research on fusion-query optimization. Our work on the simple framework of attribute adornments to describe query capabilities (presented in Chapter 5) is motivated by the use of query templates in the TSIMMIS system. We saw that the templates in TSIMMIS are rather complicated and hard to analyze and combine to arrive at the capabilities of the mediator templates. Hence, we developed the simpler framework with attribute adornments that enabled us to compute mediator capabilities easily. Thus, our work on the TSIMMIS system drove many issues whose resolution formed the basis of this thesis.

We note here that the TSIMMIS team has comprised of many Ph.D. students, faculty members and research engineers at Stanford University. The author's specific focus in the implementation of the TSIMMIS system has been in the development of the postprocessing engine in the wrapper toolkit, the plan-generator component and the execution-engine component in the mediator toolkit.

Chapter 8

Conclusion

In this thesis, we studied the query-processing challenges posed by autonomous data sources and how mediators overcome these challenges. Many of these problems emanate from the limited and diverse query capabilities of the data sources. We described the techniques we have developed to enable mediators to extend query capabilities and combine them in appropriate ways to provide seamless and powerful query interfaces to users.

We have also identified the special challenges that arise when fusing information across a large number of autonomous sources. The unpredictable data distribution across these sources makes query-optimization as well as query-processing tasks quite difficult. We presented algorithms that allow mediators to find optimal query plans involving information fusion and to execute these plans efficiently.

We created a simple but powerful framework to describe the query capabilities of data sources over the Web and developed algorithms to compute the capabilities of mediators that are built on top of Web sources. By having a framework to compute the mediator capabilities formally, we facilitate query-processing environments involving networks of mediators and sources interacting freely and exchanging queries and data according to their respective capabilities. We also developed template-minimization techniques to ensure concise query interfaces for mediators and data sources over the Web.

Finally, we presented a brief description of our research prototype TSIMMIS. In this system, we developed mediator and wrapper toolkits and have demonstrated the applicability of many of the concepts discussed in this thesis for query processing over Web sources and mediators (e.g., describing the capabilities of sources using query templates, extending query capabilities using postprocessing techniques, capability-based mediation of join queries).

We focused our attention in this thesis on the use of mediation techniques for query processing over autonomous data sources. When we consider update processing, besides the conventional challenges like the support of transactional semantics, there can be new challenges due to the autonomy of the sources in terms of limited update interfaces (like special Web forms for posting updates). Some of these new challenges can be addressed by the techniques we have developed here like decomposing a request submitted to the mediator into a sequence of requests to be sent to the sources. Just an in the case of query processing, mediators need to match the submitted update request with the update capabilities of sources and may end up processing a submitted update request over a sequence of source-update requests. Note also that the notion of translating updates on integrated views provided by a mediator into updates on the individual data sources needs us to revisit the traditional challenges of translating view updates into base updates.

The methodologies involved in update mediation may be extended to the general setting of mediating processes and/or workflow across a set of autonomous sources. These sources may have heterogeneous processes for similar tasks and they may also have limited interfaces to these processes. Mediators in future need to be actively involved in the task of managing workflows and processes across such sources so that users can be abstracted from the limited and diverse interfaces to the processes at the sources.

Another important aspect of information integration, one that we have not dealt with in this thesis, is the resolution of semantics when dealing with data from multiple sources. For instance, when a source presents information about "Java," it is not clear whether the information is referring to the Java programming language, the Java blend of coffee, or the geographical region named Java. When integrating data from this source with another source that also presents information about "Java," it is difficult to ascertain whether both the sources are referring to the same entity or two different entities. The semantics of the source data may be inferred in some situations from the meta-data about the source. Semantic resolution is also an important issue when answering user queries. For instance, when a user poses a search request looking for information related to "Java," it is once again unclear what the user is really looking for. Based on the profile of the user posing the request, the mediator may be able to infer what the user intention is. Such semantic inferencing can play a major role in ranking the relevance of the query results. Semantic inferencing also plays a significant role in fetching approximate answers to user queries in mediation environments. For instance, a digital-library source may return articles on "JDBC" (i.e., Java Database Connectivity) technologies in response to the search term "Java". We note here that we have not directly addressed semantic issues in this thesis. The work we have presented in this thesis is complementary to the techniques published in the literature on semantic resolution.

When considering mediated query processing over a large number of sources, we need to take into account the availability characteristics of the data sources. That is, in some circumstance, mediators need to be able to answer user queries even when some of the data sources are temporarily unavailable. For instance, a mediator that provides a union view over a set of sources (e.g., a meta-search engine sitting on top of a set of Web search engines) should present partial answers to user queries in the presence of source unavailability. In [84], we address the problem of how mediators choose dynamic query-execution plans that take into consideration source unavailability in a graceful manner. Our research indicates that the problem of obtaining optimal query plans in the presence of source unavailability is extremely difficult and heuristic solutions that guarantee optimality in specific scenarios need to be developed.

We conclude the thesis with a few remarks on the caching aspects of mediators. In order to improve query performance, mediators may try to cache data from the sources. They can also maintain precomputed views, like in the case of data warehouses. A pure warehouse-oriented approach has the disadvantage of potentially stale data, while promising efficient query execution and robustness with respect to data-source unavailability. On the other hand, a pure mediation-oriented approach has the disadvantage of expensive query processing and vulnerability to data-source unavailability while guaranteeing fresh answers. Depending on the application scenario at hand, we may choose one or the other approach. In many situations, the correct choice is a hybrid approach where the mediator caches some data to guarantee fast approximate query execution, while retaining the ability to provide fresh answers by consulting the data sources in real time.

Bibliography

[1] S. Abiteboul, S. Cluet, T. Milo. Correspondence and Translation for Heterogeneous Data. In *Proc. ICDT Conference*, 1997.

[2] R. Ahmed, et al. The Pegasus Heterogeneous Multidatabase System. In *IEEE Computer*, Vol. 24, 1991.

[3] P. Apers, A. Hevner, S. Yao. Optimization Algorithms for Distributed Queries. In *IEEE Transactions on Software Engineering*, 9(1), 1983.

[4] Y. Arens, C. Knoblock, W. Shen. Query Reformulation for Dynamic Information Integration. In *Journal of Intelligent Information Systems*, 6(2/3):99-130, 1996.

[5] C. Batini, M.Lenzerini, S.B. Navathe. A Comparative Analysis of Methodologies for Database Schema Integration. In *ACM Computing Surveys*, Vol. 18, 1986.

[6] P. Bernstein, N. Goodman, E. Wong, C. Reeve, J. Rothnie. Query Processing in a System for Distributed Databases (SDD-1). In *ACM Trans. Database Systems*, 6(4), 1981.

[7] J. Blakeley. Data Access for the Masses through OLE DB. In *Proc. ACM SIGMOD Conference*, 1996.

[8] P. Buneman, S. Davidson, A. Kosky. Theoretical Aspects of Schema Merging. In *Proc. EDBT Conference*, 1992.

[9] M. Carey et al. Towards Heterogeneous Multimedia Information Systems: The Garlic Approach. In *Proc. RIDE-DOM Workshop*, pages 124–131, 1995.

[10] P. Celis. The Query Optimizer in Tandem's New Serverware SQL Product. In Proc. VLDB Conference, 1996.

[11] S. Ceri, G. Gottlobb, L. Tanca. Logic Programming and Databases, Surveys in Computer Science. Springer-Verlag, 1990.

[12] C. Chang, H. Garcia-Molina. Mind Your Vocabulary: Query Mapping across Heterogeneous Information Sources. In *Data Engineering Bulletin*, 10(4):52-62, 1987.

[13] S. Chawathe, et al. The TSIMMIS Project: Integration of Heterogeneous Information Sources. In *IPSJ*, Japan, 1994.

[14] D. Chimenti, et al. An Overview of the LDL System. In *Data Engineering Bulletin*, 10(4):52-62, 1987.

[15] S. Cluet, G. Moerkotte. On the Complexity of Generating Optimal Left-deep Processing Trees with Cross Products. In *ICDT Conference*, 1995.

[16] R. Epstein, M. Stonebraker. Analysis of Distributed Database Strategies. In *Proc. VLDB Conference*, 1980.

[17] D. Florescu, et al. Query Optimization in the Presence of Limited Access Patterns. In *Proc. ACM SIGMOD Conference*, 1999.

[18] J.C. Franchitti, R. King. Amalgam: A Tool for Creating Interoperating Persistent, Heterogeneous Components. In *Advanced Database Systems*, 1993.

[19] C. Galindo-Legaria, A. Pellenkoft, M. Kersten. Fast, Randomized Join Order Selection – Why Use Transformations? In *Proc. VLDB Conference*, 1994.

[20] H. Garcia-Molina, W. Labio, R. Yerneni. Capability-Sensitive Query Processing on Internet Sources. In *Proc. ICDE Conference*, 1999.

[21] M. Garey, D. Johnson. Computers and Intractability: A Guide to the Theory of NP-Completeness. *Freeman*, 1979.

[22] P. Gassner, et al. Query Optimization in the IBM DB2 Family. In *IEEE Data Engineering Bulletin*, 16:4-18, 1993.

[23] M. Genesereth, A. Keller, O. Duschka. Infomaster: An Information Integration System. In *Proc. ACM SIGMOD Conference*, 1997.

[24] M.R. Genesereth, N.J. Nillson. Logical Foundations of Artificial Intelligence. Morgan Kauffman, 1988.

[25] G. Graefe. The Cascades Framework for Query Optimization. In Bulletin of the Technical Committee on Data Engineering, 1995.

[26] P. Gupta, E. Lin. DataJoiner: A Practical Approach to Multidatabase Access. In Proc. PDIS Conference, 1994.

[27] L. Haas, et al. Extensible Query Processing in Starburst. In *Proc. ACM SIGMOD Conference*, 1989.

[28] L. Haas, et al. Optimizing Queries Across Diverse Data Sources. In *Proc. VLDB Conference*, 1997.

[29] J. Hammer et al. Template-Based Wrappers in the TSIMMIS System. In *Proc. ACM SIGMOD Conference*, 1997.

[30] J. Hellerstein, M. Stonebraker. Predicate Migration: Optimizing Queries with Expensive Predicates. In *Proc. ACM SIGMOD Conference*, 1993.

[31] D. Hochbaum. Approximation Algorithms for Set Covering and Vertex Cover Problems. In *SIAM Journal of Computing*, 11:555-556, 1982.

[32] T. Ibaraki, T. Kameda. On the Optimal Nesting Order for Computing N-relational Joins. In *ACM Trans. Database Systems*, 9(3), 1984.

[33] Y. Ioannidis, Y. Kang. Randomized Algorithms for Optimizing Large Join Queries. In *Proc. ACM SIGMOD Conference*, 1990.

[34] Y. Ioannidis, E. Wong. Query Optimization by Simulated Annealing. In *Proc. ACM SIGMOD Conference*, 1987.

[35] O. Kapitskaia, A. Tomasic, P. Valduriez. Scaling Heterogeneous Databases and the Design of Disco. *INRIA Technical Report*, 1997.

[36] A. Kemper, et al. Optimizing Disjunctive Queries with Expensive Predicates. In *Proc. ACM SIGMOD Conference*, 1994.

[37] T. Kirk, et al. The Information Manifold. In *AAAI Symposium on Information Gathering in Distributed Heterogeneous Environment*, 1995.

[38] D. Konopnicki, O. Shmueli. W3QS: A Query System for the World-wide Web. In *Proc. VLDB Conference*, 1995.

[39] R. Krishnamurthy, H. Boral, C. Zaniolo. Optimization of Non-recursive Queries. In *Proc. VLDB Conference*, 1986.

[40] A. Levy, A. Rajaraman, J. Ordille. Querying Heterogeneous Information Sources Using Source Descriptions. In *Proc. VLDB Conference*, 1996.

[41] A. Levy, A. Rajaraman, J. Ordille. Query Answering Algorithms for Information Agents. In *Proc. AAAI Conference*, 1996.

[42] A. Levy, M. Rousset. CARIN: A Representation Language Integrating Horn Rules and Description Logics. *AT&T Technical Report*, 1995.

[43] A. Levy, Y. Sagiv. Constraints and Redundancy in Datalog. In *Proc. PODS Conference*, 1992.

[44] C. Li, et al. Capability Based Mediation in TSIMMIS. In *Proc. ACM SIGMOD Conference*, 1998.

[45] W. Litwin, L. Mark, N. Rossopoulos. Interoperability of Multiple Autonomous Databases. In *ACM Computing Surveys*, Vol. 22, No. 3, 267-293.

[46] D. Maluf, G. Wiederhold. Abstraction of Representation for Interoperation. *Lecture Notes in AI, subseries of LNCS*, Vol. 1315, 1997.

[47] G. Mecca, et al. The Araneus Web-Based Management System. In *Proc. ACM SIGMOD Conference*, 1998.

[48] W. Meng, C. Yu. Query Processing in Multidatabase Systems. In *Modern Database Systems. Addison Wesley*, 1995.

[49] R. Miller, Y. Ioannidis, R. Ramakrishnan. Schema Equivalence in Heterogeneous Systems: Bridging Theory and Practice. In *Information Systems*, 19:3-31, 1994.

[50] K. Morris. An Algorithm for Ordering Subgoals in NAIL!. In *Proc. PODS Conference*, 1988.

[51] K. Morris, J. Ullman, A. Gelder. Design Overview of the NAIL! System. In *Proc. ICLP Conference*, 1986.

[52] K. Ono, G. Lohman. Measuring the Complexity of Join Enumeration in Query Optimization. In *Proc. VLDB Conference*, 1990.

[53] T. Ozsu, P. Valduriez. *Principles of Distributed Database Systems. Prentice Hall*, 1991.

[54] Y. Papakonstantinou, S. Abiteboul, H. Garcia-Molina. Object Fusion in Mediator Systems. In *Proc. VLDB Conference*, 1996.

[55] Y. Papakonstantinou, H. Garcia-Molina, J. Ullman. Medmaker: A Mediation System Based on Declarative Specifications. In *Proc. ICDE Conference*, 1996.

[56] Y. Papakonstantinou, H. Garcia-Molina, J. Widom. Object Exchange across Heterogeneous Information Sources. In *Proc. ICDE Conference*, 251–260, 1995.

[57] Y. Papakonstantinou, A. Gupta, L. Haas. Capabilities-based Query Rewriting in Mediator Systems. In *Proc. PDIS Conference*, 1996.

[58] A. Pellenkoft, C. Galindo-Legaria, M. Kersten. The Complexity of Transformation-Based Join Enumeration. In *Proc. VLDB Conference*, 1997.

[59] A. Rajaraman, Y. Sagiv, J. Ullman. Answering Queries Using Templates With Binding Patterns. In *Proc. PODS Conference*, 1995.

[60] W. Scheufele, G. Moerkotte. On the Complexity of Generating Optimal Plans with Cross Products. In *Proc. PODS Conference*, 1997.

[61] P. Selinger, et al. Access Path Selection in a Relational Database System. In *Proc. ACM SIGMOD Conference*, 1979.

[62] P. Selinger, M. Adiba. Access Path Selection in Distributed Databases Management Systems. In *Readings in Database Systems. Morgan-Kaufman*, 1994.

[63] S. Sharma. Personal Communication with Sunil Sharma. *Tandem Computers*, 1997.

[64] A. Sheth, L. Larson. Federated Database Systems for Managing Distributed, Heterogeneous and Autonomous Databases. In *ACM Computing Surveys*, Vol. 22, No. 3, 183-236.

[65] A. Silberschatz, H. Korth, S. Sudarshan. *Database System Concepts. McGraw Hill*, 1997.

[66] M. Steinbrunn, G. Moerkotte, A. Kemper. Heuristic and Randomized Optimization for the Join Ordering Problem. In *VLDB Journal*, 6(3), 1997.

[67] M. Stonebraker. The Ingres Papers. *Addison Wesley*, 1986.

[68] V.S. Subrahmanian et al. HERMES: A Heterogeneous Reasoning and Mediator System. http://www.cs.umd.edu/projects/hermes/overview/papers. January 1998.

[69] A. Swami. Optimization of Large Join Queries: Combining Heuristic and Combinatorial Techniques. In *Proc. ACM SIGMOD Conference*, 1989.

[70] A. Swami, A. Gupta. Optimization of Large Join Queries. In *Proc. ACM SIGMOD Conference*, 1988.

[71] A. Tomasic, L. Raschid, P. Valduriez. Scaling Heterogeneous Databases and the Design of Disco. In *Proc. ICDCS Conference*, 1996.

[72] A. Tomasic, L. Raschid, P. Valduriez. Dealing with Discrepancies in Wrapper Functionality. In *Proc. ICDCS Conference*, 1996.

[73] M. Tork Roth, P. Schwarz. Don't Scrap It, Wrap It! A Wrapper Architecture for Legacy Sources. In *Proc. VLDB Conference*, 1997.

[74] S. Tsur, C. Zaniolo. LDL: A Logic-Based Data Language. In *Proc. VLDB Conference*, 1986.

[75] J. Ullman. Principles of Database and Knowledge-base Systems, Volumes I & II. *Computer Science Press*, 1989.

[76] J. Ullman, M. Vardi. The Complexity of Ordering Subgoals. In *Proc. PODS Conference*, 1988.

[77] B. Vance, D. Maier. Rapid Bushy Join-Order Optimization with Cartesian Products. In *Proc. ACM SIGMOD Conference*, 1996.

[78] V. Vassalos, Y. Papakonstantinou. Describing and Using the Query Capabilities of Heterogeneous Sources. In *Proc. VLDB Conference*, 1997.

[79] G. Wiederhold. Mediators in the Architecture of Future Information Systems. In *IEEE Computer*, 25:38-49, 1992.

[80] R. Yerneni, et al. Optimizing Large Join Queries in Mediation Systems. In *Proc. ICDT Conference*, 1999.

[81] R. Yerneni, et al. Fusion Queries over Internet Databases. In *Proc. EDBT Conference*, 1998.

[82] R. Yerneni, et al. Computing Capabilities of Mediators. In *Proc. ACM SIGMOD Conference*, 1999.

[83] R. Yerneni, H. Garcia-Molina. Design of Efficient Query Interfaces for Web Sources. *Stanford University Technical Report*, 2000. http://www-db.stanford.edu/ yerneni/pubs/deqiws.ps

[84] R. Yerneni, et al. Maximizing Coverage of Mediated Web Queries. *Stanford University Technical Report*, 2000. http://www-db.stanford.edu/ yerneni/pubs/mcmwq.ps

[85] `http://www.amazon.com/` Amazon.com Web Site. March 2001.

[86] `http://www.amazon.com/exec/obidos/ats-query-page/` Author-Title-Subject Query Form for Amazon.com. March 1999.

[87] `http://www.amazon.com/exec/obidos/subst/search/isbn.html/` ISBN Query Form for Amazon.com. March 1999.

[88] `http://www.amazon.com/exec/obidos/subst/search/publication-date.html/` Keywords-Publisher-Date Query Form for Amazon.com. March 1999.

[89] `http://autoconnect.net/FindVehicle/FVEntryPage.cfm` Search Form for Auto-Connect.net. March 1999.

[90] `http://www.barnesandnoble.com/` BarnesAndNoble.com Web Site. March 2001.

[91] `http://shop.barnesandnoble.com/booksearch/search.asp` BarnesAndNoble.com Query Forms. March 1999.

[92] `http://www.junglee.com/shop/index.html` Search Form for Junglee.com. August 1998.

[93] `http://lcweb.loc.gov/` Library of Congress Web Site. March 1999.

[94] `http://www.best.com/ sccref/catalog.html` Santa Clara Public Library Web Site. March 1999.

[95] `http://www.secapl.com/` A Web Site for Stock Quotes. March 1999.

[96] `http://www.w3.org/TR/REC-xml` Extensible Markup Language (XML) 1.0: Proposed Recommendation. May 1999.

[97] `http://shopping.yahoo.com/books` Yahoo! Shopping Guide for Books. December 1998.