MINING AND USING COVERAGE AND OVERLAP STATISTICS FOR DATA INTEGRATION

by

Zaiqing Nie

A Dissertation Presented in Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

ARIZONA STATE UNIVERSITY

May 2004

MINING AND USING COVERAGE AND OVERLAP STATISTICS

FOR DATA INTEGRATION

by

Zaiqing Nie

has been approved

March 2004

APPROVED:

____, Chair

____

____

____

____

Supervisory Committee

ACCEPTED:

____

Department Chair

____

Dean, Graduate College

ABSTRACT

Query processing in the context of integrating autonomous data sources on the Internet has received significant attention of late. In contrast to traditional query processing scenarios, in which each relation is stored in the same primary database and in which completeness of answers is expected by users, data integration scenarios involve handling relations that are stored across multiple and potentially overlapping sources and dealing with conflicting objectives in terms of what coverage of answers users want and how much execution cost they are willing to bear for achieving the desired coverage. Hence, query processing in data integration requires coverage and overlap statistics about these autonomous sources to generate optimal query plans. This dissertation first presents StatMiner, an effective statistics mining approach which automatically generates attribute value hierarchies, discovers frequently accessed query classes, and learns coverage and overlap statistics only with respect to these classes. The dissertation then introduces Multi-R, a multi-objective query optimizer which uses coverage and overlap statistics to support joint optimization of coverage and cost of query plans. The efficiency of StatMiner and the effectiveness of the learned statistics are demonstrated in the context of BibFinder, a publicly available bibliography mediator developed as a testbed for this work. The empirical evaluation of Multi-R also shows that the generated query plans are significantly better than the existing approaches, both in terms of planning cost and in terms of plan execution cost.

To my wife.

ACKNOWLEDGMENTS

First, I would like to thank my advisor Professor Subbarao Kambhampati for inspiring me to aim high and for giving me freedom to be creative. He gave freely of his time and wisdom to help me stimulate new ideas and to polish my work. I am deeply grateful to him for his constant support, guidance, and encouragement during my graduate studies in the United States.

A big thanks goes to Professor K. Selçuk Candan , Professor Huan Liu, and Professor Susan D. Urban for valuable advice and encouragement. I would specially like to acknowledge my outside committee member, Professor Louiqa Raschid of University of Maryland for giving many insightful suggestions and for taking time from her busy schedule to participate in my dissertation.

My special thanks to Thomas Hernandez, Ullas Nambiar and Sreelakshmi Vaddi for many helpful critiques and for collaborating with me on publications that are parts of the dissertation. To my friends Binh Minh Do, Romeo Sanchez Nigenda, Dan Bryce, Menkes van den Briel, Jianchun Fan, William Cushing, Dr. Terry Zimmerman, Dr. Xiaomin Li, Dr. Biplav Srivastava, Lei Yu, Le-Chi Tuan, Nam Tran, Hung. V. Nguyen, and all other members of the AI lab, thanks for their invaluable companionship.

I am and will always be very grateful to my parents who have supported me with their love and encouragement throughout my education. Last but not least, special thanks to my wife, Xiaoli Hu, who has supported me with endless love, caring, and encouragement from the beginning to the end of this work.

This research is supported in part by the NSF grant IRI-9801676 and the ASU ET-I^3 initiative grant ECR A601.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

CHAPTER 1

INTRODUCTION

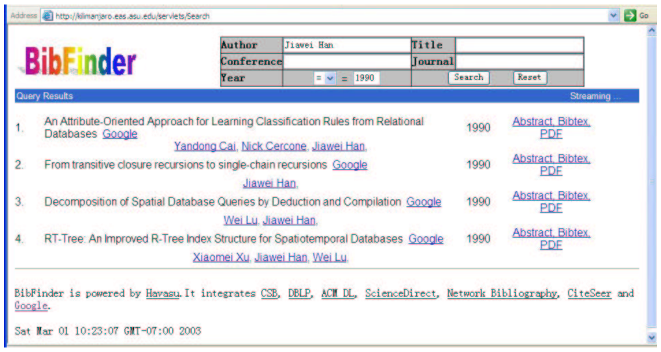
This dissertation studies how to mine and use coverage and overlap statistics about autonomous data sources to perform efficient query processing in a data integration system. We begin this chapter by showing the need for coverage and overlap statistics, and the challenges in mining and using them in query optimization. Next, we discuss the previous state of the art techniques for solving these challenges. We then outline our solution. Finally, we list our contributions and give a road map for the rest of the dissertation.

1.1. Motivation for Mining and Using Coverage and Overlap Statistics

With the vast number of autonomous information sources available on the Internet today, users have access to a large variety of data sources. Data integration systems [LRO96, DGL00, LKG99, PL00, NKH03] are being developed to provide a uniform interface to a multitude of information sources, query the relevant sources automatically and restructure the information from different sources. Query processing in the context of integrating autonomous data sources on the Internet has thus received significant attention of late. Some of the challenges in supporting such query processing – that is, the need to reformulate queries posed to the mediator into an equivalent set of queries on the data sources, the ability to handle sources with limited access capabilities, and the need to handle uncertainty during execution time, have been addressed previously in [LRO96, UF98, FLMS99, IFF$^+$99, AH00, DGL00, PL00]. There are however two critical aspects of query processing in data integration systems that have not yet been tackled adequately:

The Need for Mining Coverage and Overlap Statistics: In a data integration scenario, a user interacts with a mediator via a mediated schema. A mediated schema is a set of virtual relations, which are stored across multiple and potentially overlapping data sources, each of which may only contain a partial extension of the relation. A naive way of answering a user query would be to send it to all the mediated sources, wait for the results, eliminate duplicates, and return the answers to the user. This not only leads to increased query processing time and duplicate tuple transmission, but also unnecessarily increases the load on the individual sources. A more efficient and *polite* approach would be to direct the query only to the most relevant sources. Query processing in data integration [FKL97, NLF99, NK01, DH02] thus requires the ability to figure out what sources are most relevant to the given query, and in what order those sources should be called. For this purpose, the query optimizer needs to access statistics about the coverage of the individual sources with respect to a given query, as well as the degree to which the answers they export overlap. Given that sources tend to be autonomous in a data integration scenario, it is impractical to assume that the sources will export such statistics. Consequently, data integration systems need to be able to learn the coverage and overlap statistics about the sources they integrate.

The Need for Joint Optimization of Cost and Coverage of Query Plans: Users may have differing objectives in terms of what coverage they want to achieve and



Address http://kilimanjaro.eas.asu.edu/servlets/Search

Author	Jiawei Han	Title	
Conference		Journal	
Year	= = 1990	Search Reset	

Query Results Streaming ...

1. An Attribute-Oriented Approach for Learning Classification Rules from Relational Databases Google — 1990 — Abstract, Bibtex, PDF
   Yandong Cai, Nick Cercone, Jiawei Han,
2. From transitive closure recursions to single-chain recursions Google — 1990 — Abstract, Bibtex, PDF
   Jiawei Han,
3. Decomposition of Spatial Database Queries by Deduction and Compilation Google — 1990 — Abstract, Bibtex, PDF
   Wei Lu, Jiawei Han,
4. RT-Tree: An Improved R-Tree Index Structure for Spatiotemporal Databases Google — 1990 — Abstract, Bibtex, PDF
   Xiaomei Xu, Jiawei Han, Wei Lu,

BibFinder is powered by Havasu. It integrates CSB, DBLP, ACM DL, ScienceDirect, Network Bibliography, CiteSeer and Google.

Sat Mar 01 10:23:07 GMT-07:00 2003

Figure 1. The BibFinder user interface

how much execution cost they are willing to bear for achieving the desired coverage. Consequently, selecting high-quality plans in data integration requires the ability to consider the coverage offered by various sources, and form a query plan with the combination of sources that is estimated to be a high-quality plan given the cost-coverage tradeoffs of the user.

In the next section, we will illustrate these challenges using an example scenario.

1.2. BibFinder: Our Motivating Scenario

BibFinder Scenario: We have been developing *BibFinder* ([BF], see Figure 1), a publicly fielded computer science bibliography mediator, which integrates several online Computer Science bibliography sources. It currently covers *CSB, DBLP, Network Bibliography, ACM Digital Library, ACM Guide, IEEE Xplore, ScienceDirect,* and *CiteSeer*. Plans are underway to add several additional sources including *AMS MathSciNet* and *Computational Geometry Bibliography*. Since its unveiling in December 2002, *BibFinder* has been getting on the order of 200 queries a day.

The sources integrated by *BibFinder* are autonomous and partially overlapping. By combining the sources, *BibFinder* can present a unified and more complete view to the user. However it also brings some interesting optimization challenges. The global schema exported by *BibFinder* can be modeled in terms of the relation:

paper(title, author, conference/journal, year),

and the queries can be seen as selection queries on the paper relation. Each of the individual sources may export only a subset of the global relation. For example, *Network Bibliography* only contains publications in Networks, *DBLP* gives more emphasis to Database publications, while *ScienceDirect* has only archival journal publications.

As we discussed earlier, an efficient and polite way of answering a user query would be to direct the query only to the most relevant sources. Suppose the user asks a selection query

$$\begin{aligned}\mathbf{Q}(\text{title,author,year}) :- \; & \mathbf{paper}(\text{title, author, conference/journal, year}),\\ & \text{conference/journal} = \text{“SIGMOD”}.\end{aligned}$$

For answering this query, *DBLP*, *ACM Digital Library*, and *ACM Guide* are most relevant, while *Network Bibliography* is much less relevant. Furthermore, since *DBLP* stores records of virtually all the SIGMOD papers, a call to *ACM Digital Library* and *ACM Digital Guide* is largely redundant. In practice, *ACM Digital Library* is not completely redundant since it often provides additional information about papers – such as abstracts and citation links – that *DBLP* does not provide. *BibFinder* handles this by dividing the paper search into two phases–in the first phase, the user is given a listing of all the papers that satisfy his/her query. *BibFinder* uses a combination of three attributes: title, author, and year as the primary key to uniquely identify a paper across sources. In the second phase, the user can ask additional details on specific papers. While it is important to call every potentially relevant source in the second phase, we do not have this compulsion in the first phase. For the rest of this dissertation, all our references to *BibFinder* are to its first phase.

To judge the source relevance however, *BibFinder* needs to know the *coverage* of each source S with respect to the query Q, i.e. $P(S|Q)$, the probability that a random answer tuple for query Q belongs to source S. Given this information, we can rank all the sources in descending order of $P(S|Q)$. The first source in the ranking is the one we would want to access first while answering query Q. Since the sources may be highly correlated, after we access the source S' with the maximum coverage $P(S'|Q)$, the second source S'' that we access must be the one with the highest *residual coverage* (i.e. provides the maximum number of those answers that are not provided by the first source S'). Specifically we need to determine the source S'' that has the next best rank in terms of coverage but has minimal *overlap* (common tuples) with S'.

1.3. Challenges in Mining and Using Coverage and Overlap Statistics

If we have the coverage and overlap statistics for every possible query, we can get the complete order in which to access the sources, and select only the relevant sources to answer user queries. However, it would be prohibitively costly to learn and store statistics with respect to every source-query combination, and overlap information about every subset of sources with respect to every possible query. The difficulty here is two-fold. First, there is the cost of "learning" – which would involve probing the sources with all possible queries *a priori*, and computing the coverage and overlap with respect to the queries. Second, there is the cost of "storing" the statistics. For example, storing statistics for selection queries with respect to a single mediated relation will necessitate $N_q * 2^{N_S}$ different statistics, where N_q is the number of different selection queries on the mediated relation and N_S is the number of mediated sources. Since both the number of possible queries[1] and the number of sources

[1]Although in real data integration scenarios, it is impractical to assume we can know the entire query population *a priori*.

exporting the relation could be very large, it is impractical to keep all the statistics in the main memory of a mediator.

Once we have the coverage and overlap statistics, effectively using them in query optimization also presents several challenges. If we want to get a query plan that can cover more relevant answers with limited cost, it is critical to consider execution costs *while* doing source selection (rather than *after the fact*). In order to take the cost information into account, we have to consider the source-call ordering during planning, since different orders will result in different execution costs for the same logical plan. However, the search space of all query plans can be prohibitively large. In particular, for a join query with n subgoal relations, and m sources exporting each subgoal relation, the size of the search space is $2^{m \times n} \times n!$, as there can be 2^m source combinations for each subgoal, hence $2^{m \times n}$ possible logical query plans, and the number of subgoal orders for each plan is $n!$.

1.4. State of the Art

The utility of quantitative coverage statistics to rank the sources was first explored by Florescu *et. al.* [FKL97]. However the primary aim of the effort was to model the coverage and overlap statistics, and it did not discuss how such statistics could be learned. The work introduced two simple algorithms to use coverage and overlap statistics to rank mediated sources, however they did not address how these statistics can be used along with other types of statistics (such as response time) in query optimization. Thus far there has not been any work on effectively learning the statistics in the first place.

More recent work [NLF99; DH02] tries to use coverage and overlap statistics together with other types of statistics (including response time) for optimizing queries in data integration. However they use decoupled strategies – attempting to optimize coverage and cost in two separate phases. Specifically, they first generate a set of feasible “linear plans” that contain at most one source for each query conjunct, and then rank these linear plans in terms of the expected coverage offered by them. Finally, they select the top N plans from the ranked list and execute them. Since sources tend to have a variety of access limitations, this type of phased optimization of cost and coverage can unfortunately lead to significantly costly planning *and* inefficient plans.

1.5. Overview of our Solution

This dissertation introduces a novel query processing framework in which we adapt data mining techniques to automatically gather coverage and overlap statistics about the mediated sources, and use the gathered statistics to support multi-objective query optimization. Specifically, my dissertation introduces *StatMiner*, a statistics mining module for efficient query processing in a data integration system. The purpose of *StatMiner* is to group queries into classes and to efficiently discover frequent query classes for which we would like to store statistics. It starts with a list of user queries which can be collected from the log of queries submitted to a mediator. The query list not only gives the specific queries submitted to the mediator, but also coverage and overlap statistics on how many tuples for each query came from which source. For mediators in their beginning stages, a query list can be generated by probing the mediated sources. The query list is used to automatically generate attribute value hierarchies[2] which are then employed to form query classes. In addition, the query list is used to prune query classes that subsume less than a given number of user queries (specified by a frequency threshold). For each of the remaining classes, coverage and overlap statistics are learned.

[2] An *attribute value hierarchy* over an attribute in a mediated relation is a hierarchical classification of the values of the attribute. See Section 3.2 for a detailed discussion.

We use these statistics to select relevant sources in query optimization. Specifically, we discuss how the learned statistics can be used to estimate the coverage and overlap of the sources for a new user query, and present *Multi-R*, a multi-objective query optimizer which supports joint optimization of coverage and cost of query plans using coverage and overlap statistics. *Multi-R* searches in the space of "parallel" query plans that support parallel access to multiple sources for each subgoal conjunct. The parallel plans are evaluated in terms of a general "utility" metric, combining both cost and coverage of the plan. The plan generation process takes into account the potential parallelism between source calls, and the binding compatibilities between the sources. It includes two interleaved processes: the first is a search conducted in the space of subgoal orders, and the second is a greedy procedure that provides a high-quality subplan for a given subgoal.

The efficiency of *StatMiner* and the effectiveness of the learned statistics in selecting relevant sources are demonstrated in the context of *BibFinder*, a publicly available bibliography mediator we developed as a testbed for this work. *Multi-R* is evaluated using simulated source statistics, and the experimental results show that the generated query plans are significantly better than the existing approaches, both in terms of planning cost and in terms of plan execution cost.

1.6. Contributions of the Dissertation

Up to the time of this dissertation, most works have concentrated on only how to use coverage and overlap statistics using decoupled strategies. There has been no work on gathering these statistics, and the only work [FLK97] which discusses how to model coverage and overlap statistics requires a topic hierarchy. However the manual creation of topic hierarchies is known to be laborious and error-prone. The main contributions of the dissertation are:

- A model for supporting a hierarchical classification of the set of queries. We introduce the concept of attribute value hierarchy, and a method to automatically generate these hierarchies. Using the learned hierarchies, user queries can be conveniently classified into classes without requiring manual intervention.

- A frequency-based approach for dynamically controlling the resolution of the learned statistics. In order to keep both learning and storage costs down, we learn statistics only with respect to a smaller set of "frequently asked" query classes. This strategy trades accuracy of statistics for reduced statistics learning/storing costs. The motivation for such an approach is that it is impractical for a mediator to provide accurate statistics for every possible query, but it can still achieve a reasonable average accuracy by keeping more accurate statistics for frequent queries, and less accurate statistics for infrequent queries. The effectiveness of this approach is predicated on the belief that in real-world scenarios, the distribution of queries posed to a mediator is not *uniform*, but rather Zipfian [Zipf29] (See Section 3.1 for a detailed discussion).

- A size-based approach to provide statistics for mediators at the beginning stages. For mediators without the knowledge of user query distributions, the initial statistics are learned based on the assumption that large query classes will be accessed more frequently.

- A joint query optimization approach to optimize parallel query plans in terms of both coverage and cost. We introduce ways in which cost and coverage of query plans can be estimated and combined into an aggregated utility measure. The parallel plan generation process takes into account the potential parallelism between source calls, and the binding compatibilities between the sources included in the plan. An important advantage of parallel plans over linear plans is that they avoid the significant redundant computation inherent in executing all feasible linear plans separately. Our approach involves searching in the space of subgoal orders, and for each subgoal order efficiently generating a high-quality parallel plan. It winds up adding very little additional planning overhead over that of searching in the space of linear plans, and even this overhead is more than made up for by the fact that we avoid the inefficiencies of phased optimization.

- *BibFinder* – An effective testbed for data integration research. We consider our *BibFinder* testbed itself as a contribution because of the perennial problem of lack of effective testbeds for data integration research as pointed out by leading researchers in the community in the Lowell Report [Lowell03].

1.7. Outline of the Dissertation

The rest of the dissertation is organized as follows. Chapter 2 provides background about data integration systems and introduces the basic components of a data integration system. In Chapter 3, we describe *StatMiner* in detail. Specifically we present algorithms to automatically learn attribute value hierarchies and to discover frequent query class. We also present extensive experiments performed on both *BibFinder* and a controlled dataset. Chapter 4 presents *Multi-R*. We describe the models for estimating the cost and coverage of parallel plans and present two algorithms to generate parallel query plans and analyze their complexity. We also present a comprehensive empirical evaluation which demonstrates that

Multi-R can offer high utility plans (in terms of cost and coverage) for a fraction of the planning cost incurred by the existing approaches that use phased optimization with linear plans. Chapter 5 then discusses related work. Chapter 6 concludes and presents several important directions in which my dissertation work could be extended.

CHAPTER 2

BACKGROUND: DATA INTEGRATION SYSTEMS

In this chapter we will introduce the architecture of a data integration system, and briefly summarize the existing work on data integration systems.

2.1. System Overview

Data integration systems provide their users with a virtual *mediated schema* to query over. This schema is a uniform set of relations serving as the application's ontology and is used to provide the user with a uniform interface to a multitude of heterogeneous data sources, which store the actual available data. We model the contents of these data sources with a set of *source relations* which are defined as views over the mediated schema relations.

2.1.1. Architecture. In Figure 2, we show the architecture of a data integration system. In a data integration system, the user queries asked via the mediated schema are reformulated into queries over the integrated data sources. The query optimizer will find a high-quality query plan using the statistics learned by the statistics miner. The query executor will then execute the plan by calling the wrappers for the integrated data sources. We will explain each of these components in detail in the following sections.

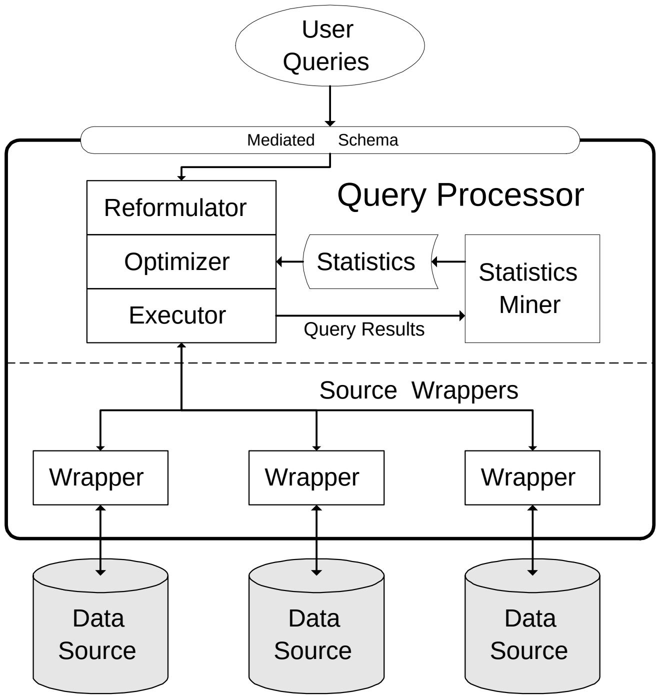


Figure 2. Data integration system architecture

2.1.2. Application Scenarios. Two types of application scenarios have been considered in the literature:

- Enterprise-oriented mediators: Integrating a set of heterogeneous database systems owned by a given corporation/enterprise;
- Domain-oriented mediators: Integrating a set of data sources that export information related to some specific application domain (e.g. BookFinder (www.boobfinder.com): comparison shopping of books, *BibFinder*: integration of multiple bibliography sources etc.).

In the first type of application scenario, we would expect that the set of data sources is relatively stable, and that the mediation is “authorized” (in that the data sources are aware that they are being integrated). In domain-oriented mediators, the set of data sources may be changing, and more often than not, the mediation may not have been explicitly authorized by the sources. Systems such as Garlic [HKWY97], TSIMMIS [CGMH94], and HERMES [ACPS96] can be characterized as aiming at enterprise-oriented applications, while InfoMaster [GGKS95], Information Manifold [LRO96], Occam [KW96], Razor [FW97], DISCO [TRV98], *Emerac* [LKG99, KLN^{+}04] as well as the *BibFinder* system [NKH03] address the domain-oriented applications. My dissertation concentrates on domain-oriented mediators, although much of the work can also be applied to enterprise-oriented mediators.

2.2. Query Reformulator

There are two broad approaches for modeling source and mediator schemas [Lev00]. The "global as view" (GAV) approach involves modeling the mediator schema as a view on the (union of) source schemas. The "local as view" (LAV) approach involves modeling the sources schemas as views on the mediated schema. The GAV approaches make query planning relatively easy as a query posed on the mediated schema can directly be rewritten in terms of the source schemas. The LAV approach, in contrast, would require a more complex rewriting phase to convert the query posed on the mediator schema to a query on the sources. The advantage of the LAV approach however is that adding new sources to a mediator involves simply modeling them as views on the mediated schema, while in the GAV approach, the mediated schema has to be rewritten every time a new source is added. Systems that address integration in the context of enterprise-oriented applications, such as Garlic, TSIMMIS and HERMES use the GAV approach as they can be certain of a relatively stable set of sources. In contrast, systems aimed at domain-oriented applications, such as the Information Manifold InfoMaster, *Emerac*, and *BibFinder* use the LAV approach.

In the LAV approach, the mediator builds a virtual global schema for the information that the user is interested in, and describes the accessible information sources as materialized views on the global schema (see [LK97]). The user query is posed in terms of the relations of the global schema. Since the global schema is virtual (in that its extensions are not stored explicitly anywhere), computing the query requires rewriting (or "folding" [Qia96]) the query such that all the EDB predicates in the rewrite correspond to the materialized view predicates that represent information sources. Several researchers [LRO96,Qia96,KW96,DG97,DL97] have addressed this rewriting problem.

2.3. Query Optimizer and Statistics Miner

The other main difference among information integration systems is the types of approaches used for query optimization. Systems addressing integration in enterprise-oriented applications can count on the availability of response time related statistics on the databases (sources) being integrated. Thus Garlic, TSIMMIS, and HERMES systems attempt to use parameterized decision model for query planning. Systems addressing domain-oriented applications, on the other hand, cannot count on having access to statistics about the information sources. Thus, either the mediator has to learn the statistics it needs, or will have to resort to optimization algorithms that are not dependent on complete statistics about sources. Both Infomaster and Information Manifold system use heuristic techniques for query optimization.

My dissertation assumes that a query optimizer uses a parameterized decision model with statistics which were gathered by the mediator using a statistics mining component. There has been some previous work on learning database statistics both in multi-database literature and data integration literature. Much of it, however, focused on learning response time statistics. Zhu and Larson [ZL96] describe techniques for developing regression cost models for multi-database systems by selective querying. Adali *et. al.* [ACPS96] discuss how keeping track of rudimentary access statistics can help in doing cost-based optimizations. More recently, the work by Gruser *et. al.* [GRZ$^+$00] considers mining response time statistics for sources in data integration scenarios, and the work by Zadorozhny *et. al.* [ZRV$^+$02] discuss how to use mined response time statistics in data integration. In contrast, my dissertation focuses on learning coverage and overlap statistics. As has been argued by us [NK01] and others [DH02], query optimization in data integration scenarios requires both types of statistics.

2.4. Query Executor and Wrapper

The query plan generated by the optimizer is executed by the query executor. Because of the high possibility of unanticipated delays and failures in answering queries using remote data data sources, several adaptive query execution engines such as Tukwila [IFF$^+$99], Query Scrambling [UFA98, UF00], and Eddies [AH00, MSHR02] have been developed to provide steady performance by re-optimizing query plans during execution.

Since the integrated data sources may be heterogenous, wrappers are developed for each of the sources to provide a uniform query interface for the query executor. For each integrated data source, a wrapper is developed to accept queries from the mediator and retrieve the answers from the source. Both the queries sent to each wrapper and the answers returned by each wrapper follow a uniform format. Although wrapper generation could become much easier as the Internet moves towards XML and Web Services, many of the current data sources still only support HTML. Much research has been done in building wrappers for online sources. Both supervised machine learning techniques [KWD97, HD98, BFG01, MMK01] and unsupervised grammar induction techniques [CMM01, HC01, LKM01] have been proposed to automatically build wrappers. To ensure the wrappers continue to work properly over time, several wrapper maintenance techniques [Kus00, LMK03] have also been introduced.

CHAPTER 3

MINING COVERAGE AND OVERLAP STATISTICS

In this chapter, we motivate and investigate the issues involved in statistics gathering in a data integration system.

3.1. Motivation

As we discussed earlier, it's impractical to learn and store the coverage and overlap statistics for every possible query *a priori*. One way of keeping both learning and storage costs down is to learn statistics only with respect to a smaller set of "frequently asked" queries that are likely to be most useful in answering user queries. This strategy trades accuracy of statistics for reduced statistics learning/storing costs. In the *BibFinder* scenario, for example, we could learn statistics with respect to the list of queries that are actually posed to the mediator over a period of time. The motivation for such an approach is that even if a mediator cannot provide accurate statistics for every possible query, it can still achieve a reasonable average accuracy by keeping more accurate coverage and overlap statistics for queries that are asked more frequently, and less accurate statistics for infrequent queries. The effectiveness of this approach is predicated on the belief that in most real-world scenarios, the distribution of queries posed to a mediator is not *uniform*, but rather

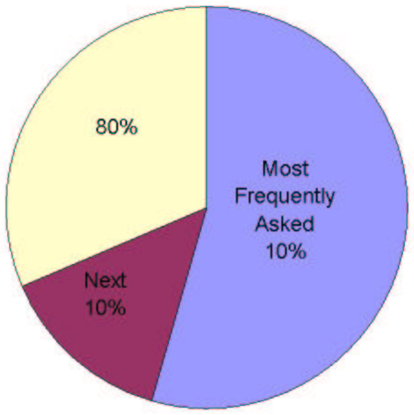


Figure 3. Keyword distribution in *BibFinder* user queries

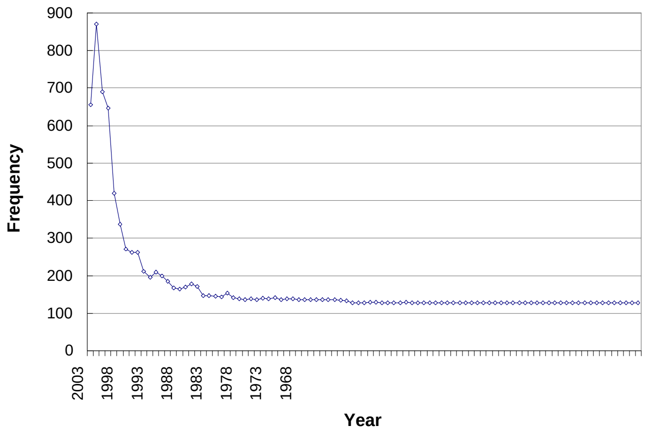


Figure 4. Year distribution in *BibFinder* user queries

Zipfian[1]. This belief is amply validated in *BibFinder*. Figure 3 shows the distribution of the keywords, and bindings for the Year attribute used in the first 15000 queries that were posed to *BibFinder*. Figure 3 shows that the most frequently asked 10% keywords appear in almost 60% of all the selection queries binding attribute Title. Figure 4 shows that the users are much more interested in recently published papers.

[1]A distribution of probabilities of occurrence that follows Zipf's law. According to Zipf's law, the probability of occurrence of words or other items starts high and tapers off. Thus, a few occur very often while many others occur rarely.

Handling New Queries through Generalization: Once we subscribe to the idea of learning statistics with respect to a workload query list, it would seem as if the problem of statistics gathering is solved. When a new query is encountered, the mediator simply needs to look into the query list to see the coverage and overlap statistics on this query when it was last executed. In reality, we still need to address the issue of what to do when we encounter a query that was not covered by the query list. The key here is "generalization"– store statistics *not* with respect to the specific queries in the query list, but rather with respect to query classes. The query classes will have a general-to-specific partial ordering among them. This in turn induces a hierarchy among the query classes, with the query list queries making up the leaf nodes of the hierarchy. The statistics for the general query classes can then be computed in terms of the statistics of their children classes. When a new query is encountered that was not part of the workload query list, it can be mapped into the set of query classes in the hierarchy that are most similar, and the (weighted) statistics of those query classes can be used to handle the new query. Such an organization of the statistics offers an important additional flexibility: we can limit the amount of statistics stored as much as we desire by stripping off (and not storing statistics for) parts of the query hierarchy.

Modeling Query Classes: The foregoing discussion about query classes raises the issue regarding the way query classes are defined. For selection queries that bind (a subset of) attributes to specific values (such as the ones faced by *BibFinder*), one way is to develop "general-to-specific" hierarchies over attribute values (AV hierarchies, see below). The query classes themselves are then naturally defined in terms of (cartesian) products over the AV hierarchies. Figure 6 shows an example of AV hierarchies and the corresponding query classes (see Section 3.2 for details). An advantage of defining query classes through the cartesian product of AV hierarchies is that mapping new queries into the query class hierarchy is straightforward – a selection query binding attributes A_i and A_j will only be mapped to a query class that binds either one or both of those attributes (to possibly general values of the attribute).[2]

Large versus Frequent Query Classes: In the above discussion, we have assumed that the mediator will maintain a query list. However the query list may not be available for mediators at their beginning stages. For such cases we introduce a size-based approach to learning statistics. There we assume that query classes with more answer tuples will be accessed more frequently, and learn coverage statistics with respect to large query classes. For example, in the *BibFinder* scenario, if the number of papers for a conference is large, we assume the *BibFinder* users will be more interested in this conference than some small conferences.[3] In the size-based approach, the query list is generated by probing the sources, and the number of the answer tuples of a query is used as the frequency of the query.

[2]This also explains why we don't cluster the query list queries directly–there is no easy way of deciding which query cluster(s) a new query should be mapped to without actually executing the new query and using its coverage and overlap statistics to compute the distance between that query and all the query clusters!

[3]The reason why good conferences usually are large is that they usually exist longer. If a conference has been held for 30 years, then the number of the papers published by the conference will usually be larger than that by a conference with only several year history.

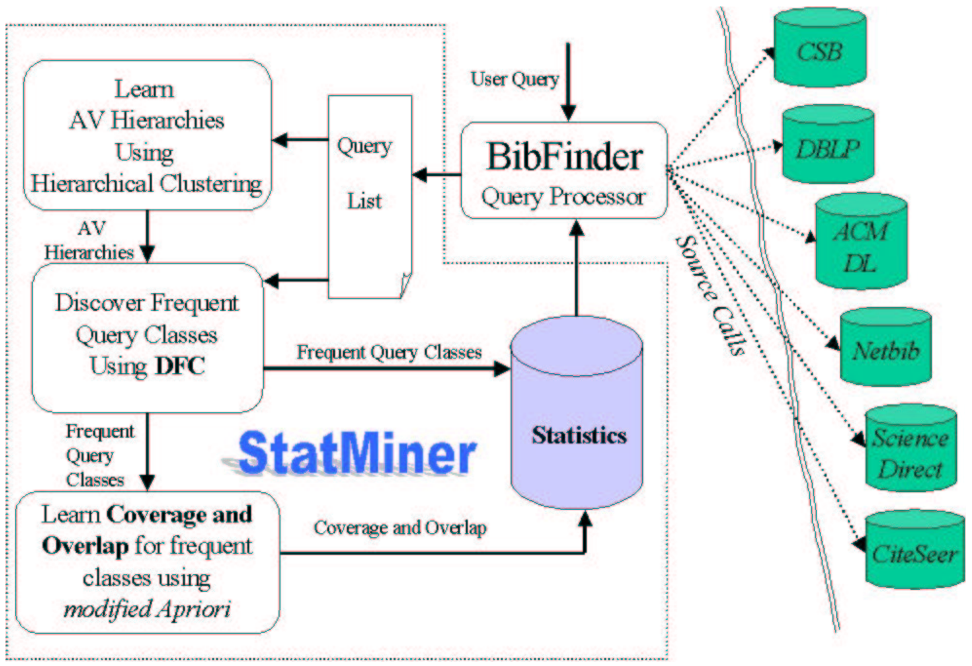

Learn AV Hierarchies Using Hierarchical Clustering
Query List
User Query
BibFinder Query Processor
AV Hierarchies
Discover Frequent Query Classes Using DFC
Frequent Query Classes
Statistics
StatMiner
Frequent Query Classes
Learn Coverage and Overlap for frequent classes using modified Apriori
Coverage and Overlap
Source Calls
CSB
DBLP
ACM DL
Netbib
Science Direct
CiteSeer

Figure 5. StatMiner architecture

The approach to statistics learning described and motivated in the foregoing has been implemented in *StatMiner*, and has been evaluated in the context of *BibFinder*. Figure 5 shows the high-level architecture of *StatMiner*. *StatMiner* starts with a list of user queries. The query list can be collected from the log of queries submitted to *BibFinder*, and not only gives the specific queries submitted to *BibFinder*, but also coverage and overlap statistics on how many tuples for each query came from which source. Initially query lists can also be generated by probing the integrated sources. The query list is used to learn AV hierarchies, and to prune query classes that subsume less than a given number of user queries (specified by a frequency threshold). For each of these remaining classes, class-source as well as class-source set association rules are learned. An example of a class-source association rule could be that $SIGMOD \rightarrow DBLP$ with confidence 100%, which means that the information source $DBLP$ covers all the paper information for $SIGMOD$ related queries.

The rest of the chapter is organized as follows. In the next section, we introduce our hierarchical query classification model. Section 3.3 describes the methodology used for extracting and processing training data from autonomous Web sources. Section 3.4 describes the details of learning AV hierarchies. Section 3.5 describes how query classes are formed. Section 3.6 describes how coverage and overlap statistics are learned for the query classes that are retained. Section 3.7 describes how a new query is mapped to the appropriate query classes, and how the combined statistics are used to develop a query plan. Section 3.8 describes the setting for the experiments we have done with *StatMiner* and *BibFinder* to evaluate the effectiveness of our approach. Section 3.9 presents the experimental results. Section 3.10 discusses how to learn coverage and overlap statistics for join queries, and Section 3.11 presents the summary.

3.2. AV Hierarchies and Query Classes

To better illustrate the novel aspects of *StatMiner*, we purposely limit the queries to just projection and selection queries. See Section 3.10 for a discussion on how our techniques can be extended to handle join queries.

AV Hierarchy: As we consider selection queries, we can classify the queries in terms of the selected attributes and their values. To abstract the classes further we assume that the mediator has access to the so-called “attribute value hierarchies” for a subset of the attributes of each mediated relation. An *AV hierarchy* (or attribute value hierarchy) over an attribute A is a hierarchical classification of the values of the attribute A. The leaf nodes of the hierarchy correspond to specific concrete values of A, while the non-leaf nodes are abstract values that correspond to the union of values below them. Figure 6 shows two very simple AV hierarchies for the "conference" and "year" attributes of the "paper" relation. Note that the hierarchies do not have to exist for every attribute, but rather only for those attributes over which queries are classified. We call such attributes the **classificatory attributes**. In the dissertation, we assume that the selection of the classificatory attributes will be done by the mediator designer,[4] although it can also be done using automated feature selection techniques. Similarly, the AV hierarchies themselves can either be hand-coded by the designer, or can be learned automatically. In Section 3.4, we give details on how we learn them automatically.

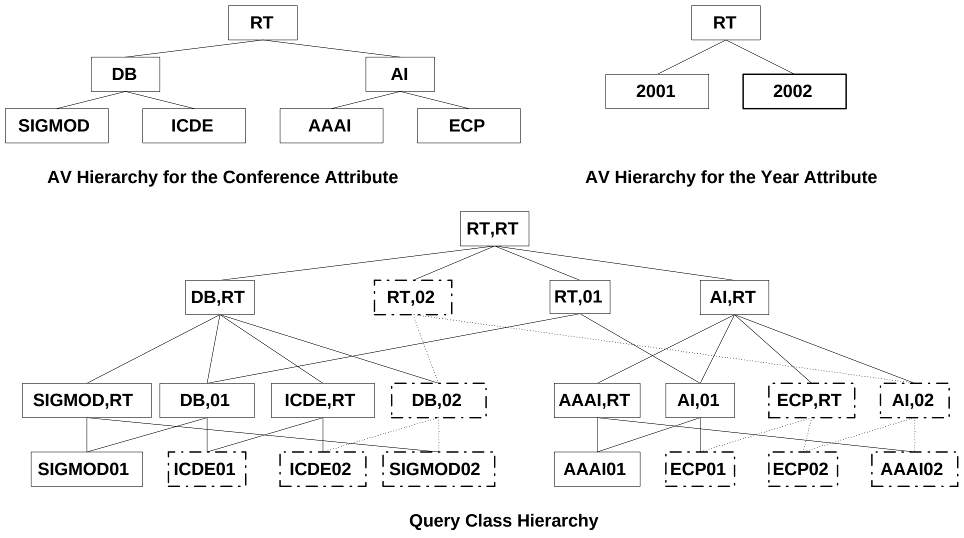


Figure 6. AV hierarchies and the corresponding query class hierarchy

Query Classes: Since a typical selection query will have values of some set of attributes bound, we group such queries into query classes using the AV hierarchies of the classificatory attributes. A query **feature** is defined as the assignment of a classificatory attribute to a specific value from its AV hierarchy. A feature is "abstract" if the attribute is assigned

[4]In scenarios with small number of attributes, we can consider all attributes as classificatory attributes. For example, in our *BibFinder* experiments, we consider all four attributes as classificatory attributes and automatically learn AV hierarchies for them.

an abstract (non-leaf) value from its AV hierarchy. Sets of features are used to define query classes. Specifically, a query class is a set of (selection) queries that all share a particular set of features. A query class having no abstract features is called a *leaf class*, similarly a query having concrete features for all the classificatory attributes is called a *leaf query*. The space of query classes is just the cartesian product of the AV hierarchies of all the classificatory attributes. Specifically, let H_i be the set of features derived from the AV hierarchy of the i^{th} classificatory attribute. Then the set of all query classes (called $classSet$) is simply $H_1 \times H_2 \times ... \times H_n$. The AV hierarchies induce subsumption relations among the query classes. A class C_i is subsumed by class C_j if every feature in C_i is equal to, or a specialization of, the same dimension feature in C_j. A query Q is said to belong to a class C if the values of the classificatory attributes in Q are equal to, or are specializations of, the features defining C. Figure 6 shows an example class hierarchy for a very simple mediator with two example AV hierarchies. The query classes are shown at the bottom, along with the subsumption relations between the classes.

Query Probability and Class Probability: For scenarios where user query distribution is available, we use FR_Q to denote the access frequency of a query Q, and FR to denote the total frequency of all the queries in the query list. The *query probability* of a query Q, denoted by $P(Q)$, is the probability that a random query posed to the mediator is the query Q, and is estimated as: $P(Q) = \frac{FR_Q}{FR}$. The *class probability* of a query class C, denoted by $P(C)$, is the probability that a random query posed to the mediator is subsumed by the class C. It is computed as: $P(C) = \sum_{Q \in C} P(Q)$.

For scenarios where user query distribution is not available, we use the number of answers of the a query (or a query class) to represent the access frequency of the query (or the query class). We use N_Q to denote the number of answers of a query Q, and N to denote the total number of distinct answers of all the leaf queries which are submitted to *BibFinder* as probing queries. The the query probability is estimated as: $P(Q) = \frac{N_Q}{N}$, and the class probability is estimated as: $P(C) = \sum_{Q \in C} P(Q)$.[5]

Coverage and Overlap w.r.t Query Classes: The *coverage* of a data source S with respect to a query Q, denoted by $P(S|Q)$, is the probability that a random answer tuple of query Q is present in source S. The *overlap* among a set $\widehat{S}$ of sources with respect to a query Q, denoted by $P(\widehat{S}|Q)$, is the probability that a random answer tuple of the query Q is present in each source $S \in \widehat{S}$. The overlap (or coverage when $\widehat{S}$ is a singleton) statistics w.r.t. a query Q are computed using the following formula

$$P(\widehat{S}|Q) = \frac{N_Q(\widehat{S})}{N_Q}$$

Here $N_Q(\widehat{S})$ is the number of answer tuples of Q that are in all sources of $\widehat{S}$, N_Q is the total number of answer tuples for Q. We assume that the union of the contents of the available sources within the system covers 100% of the answers of the query. In other words, coverage and overlap are measured relative to the available sources.

We also define coverage and overlap with respect to a query class C rather than a single query Q. The overlap of a source set $\widehat{S}$ (or coverage when $\widehat{S}$ is a singleton) w.r.t. a query class C can be computed using the following formula:

$$P(\widehat{S}|C) = \frac{P(C \cap \widehat{S})}{P(C)} = \frac{\sum_{Q \in C} P(\widehat{S}|Q)P(Q)}{P(C)}$$

The coverage and overlap statistics w.r.t. a class C is used to estimate the source coverage and overlap for all the queries that are mapped into C. Here we use the independence

[5]Note that in the size-based approach we only consider leaf queries within a query class to compute the statistics of their ancestor classes since the non-leaf queries will only have answers of leaf queries subsumed by them.

assumption: the queries within a query class are asked independently. These statistics can be conveniently computed using an association rule mining approach as discussed below.

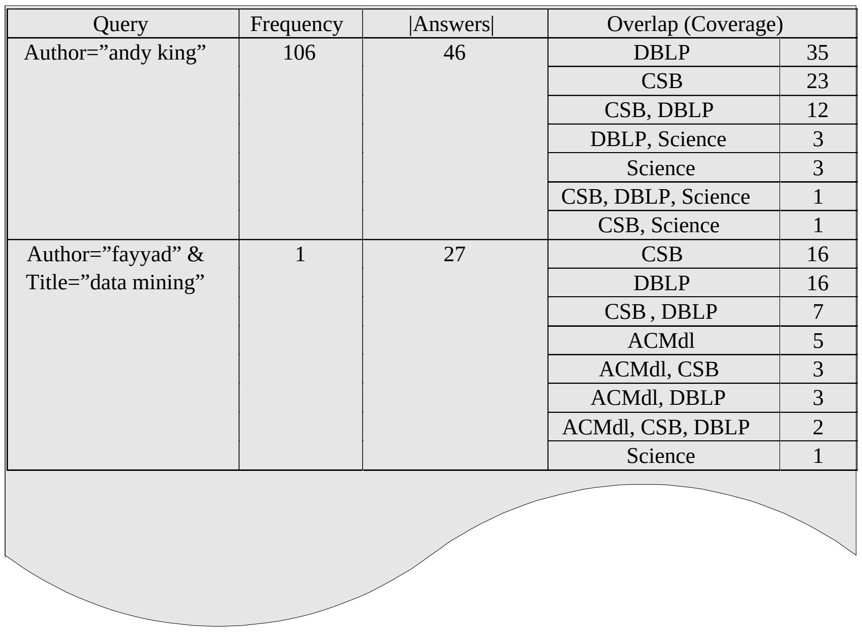
Class-Source Association Rules: A *class-source association rule* represents strong associations between a query class and a source set (which is some subset of sources available to the mediator). Specifically, we are interested in the association rules of the form $C \rightarrow \widehat{S}$, where C is a query class, and $\widehat{S}$ is a source set (possibly singleton). The *support* of the class C (denoted by $P(C)$) refers to the class probability of the class C, and the overlap (or coverage when $\widehat{S}$ is a singleton) statistic $P(\widehat{S}|C)$ is simply the *confidence* of such an association rule (denoted by $P(\widehat{S}|C) = \frac{P(C \cap \widehat{S})}{P(C)}$). Examples of such association rules include: $AAAI \rightarrow S_1$, $AI \rightarrow S_1$, $AI\&2001 \rightarrow S_1$ and $2001 \rightarrow S_1 \wedge S_2$.

3.3. Gathering Base Data

To use the association rule mining approach to learn the coverage and overlap statistics, we need to first collect a representative sample of the data stored in the sources. One way of getting the sample data is to use a query list to remember the queries submitted by users and the data distribution over the sources for these queries. In the beginning stages, however, the mediator may not have a sufficiently large and representative query log. In such scenarios, another way is to probe the sources with a representative set of probing queries. In this section, we will first explain the concept of Query List, and then discuss how to select probing queries to generate an initial query list at the beginning stages of a mediator.

3.3.1. Query List.

We assume that the mediator maintains a query list *QList*, which keeps track of the user queries, and for each query saves statistics on how often it is asked and how many of the query answers came from which sources. In Figure 7, we



Query	Frequency	|Answers|	Overlap (Coverage)	
Author=”andy king”	106	46	DBLP	35
			CSB	23
			CSB, DBLP	12
			DBLP, Science	3
			Science	3
			CSB, DBLP, Science	1
			CSB, Science	1
Author=”fayyad” & Title=”data mining”	1	27	CSB	16
			DBLP	16
			CSB , DBLP	7
			ACMdl	5
			ACMdl, CSB	3
			ACMdl, DBLP	3
			ACMdl, CSB, DBLP	2
			Science	1

Figure 7. A query list fragment

show a query list fragment. The statistics we remember in the query list are: (1) the query frequency, (2) the total number of distinct answers from all the sources, and (3) the number of answers from each source set which has answers for that query. The query list is kept as a XML file which can be stored on the mediator's hard disk or other separate storage devices. Only the learned statistics for the frequent query classes will remain in the mediator's main memory for fast access.

3.3.2. Probing Queries. There are two possible ways of generating “representative” probing queries. We could either (1) pick our sample of queries from a set of “spanning queries”–i.e., queries which together cover all the tuples stored in the data sources or (2) pick the sample from the set of actual queries that are directed at the mediator over a period of time. Although the second approach is more sensitive to the actual queries that are encountered, it has a “chicken-and-egg” problem as no statistics can be learned until the mediator has processed a sufficient number of user queries. For the purposes of the size-based approach, we shall assume that the probing queries are selected from a set of spanning queries.

For scenarios where AV hierarchies are available, spanning queries can be generated by considering a cartesian product of the leaf node features of all the classificatory attributes (for which AV hierarchies are available), and generating selection queries that bind attributes using the corresponding values of the members of the cartesian product. Every member in the cartesian product is a "least general query" that we can generate using the classificatory attributes and their AV-hierarchies. Given multiple classificatory attributes, such queries will bind more than one attribute and hence we believe they would satisfy the "binding restrictions" imposed by most autonomous Web sources. Although a query binding single classificatory attribute will generate larger result sets, most often such queries will not satisfy the binding restrictions of Web sources as they are too general and may extract a large part of the source's data. The "less general" the query (more attributes bound), the more likely it will be accepted by autonomous Web sources. But reducing the generality of the query does entail an increase in the number of spanning queries leading to larger probing costs if sampling is not done.

For scenarios where AV hierarchies are not available, we need to gather valid binding values for the attributes in the mediated relation to generate selection queries as spanning queries. For example, in our *BibFinder* scenario, we can get author names from CiteSeer and conference names from DBLP. We also know the values for the year attribute are in 1950-2004.

Once we decide the space from which the probing queries are selected (in our case, a set of spanning queries), the next question is how to pick a representative sample of these queries. Clearly, sending all potential queries to the sources is too costly. We use sampling techniques for keeping the number of probing queries under control. Two well-known sampling techniques are applicable to our scenario: (a) *Simple Random Sampling* and (b) *Stratified Random Sampling* [C77]. Simple random sampling gives equal probability of being selected to each query in the collection of sample queries. Stratified random sampling requires that the sample population be divisible into several subgroups. Then for each subgroup a simple random sampling is done to derive the samples. If the strata are selected intelligently, stratified sampling gives statistics with higher precision than simple random sampling. We evaluate both these approaches experimentally to study the effect of sampling on our learning approach.

Once we decide on a set of sample probing queries, these queries are submitted to all the data sources. The results returned by the sources are then organized in a query list (see Section 3.3.1), and the numbers of the answers of the queries are used as the access frequencies.

3.4. Generating AV Hierarchies Automatically

In this section we discuss how to systematically build AV Hierarchies based on the query list maintained by the mediator. We first define the distance function between two attribute values. Next we introduce a clustering algorithm to automatically generate AV Hierarchies. Then we discuss some complications of the basic clustering algorithm: preprocessing different types of attribute values from the query list and estimating the coverage and overlap statistics for queries with low selectivity binding values. Finally we discuss how to flatten our automatically generated AV Hierarchies.

Distance Function: The main idea of generating an AV hierarchy is to cluster similar attribute values into classes in terms of the coverage and overlap statistics of their corresponding selection queries binding these values. The problem of finding similar attribute values becomes the problem of finding similar selection queries. In order to find similar queries, we define a distance function to measure the distance between a pair of selection queries (Q_1, Q_2):

$$d(Q_1, Q_2) = \sqrt{\sum_i [P(\widehat{S}_i|Q_1) - P(\widehat{S}_i|Q_2)]^2}$$

Where $\widehat{S}_i$ denotes the i^{th} source set of all possible source sets in the mediator. Although the number of all possible source sets is exponential in terms of the number of available sources, we only need to consider source sets with answers for at least one of the two queries to compute $d(Q_1, Q_2)$.[6] Note that we are not measuring the similarity of the answers of Q_1 and Q_2, but rather the similarity of the way their answer tuples are distributed over the sources. In this sense, we may find that a selection query $conference$ = "*AAAI*" and another query $conference$ = "*SIGMOD*" to be similar in as much as the sources having tuples for the former also have tuples for the latter. Similarly we define a distance function to measure the distance between a pair of query classes (C_1, C_2):

$$d(C_1, C_2) = \sqrt{\sum_i [P(\widehat{S}_i|C_1) - P(\widehat{S}_i|C_2)]^2}$$

We compute a query class's coverage and overlap statistics $P(\widehat{S}|C)$ according to the definition of the overlap (or coverage) w.r.t. to a class given in Section 3.2. The statistics $P(\hat{S}|Q)$ for a specific query Q are computed using the statistics from the query list maintained by the mediator.

[6]For example, suppose query Q_1 gets tuples form only sources S_1 and S_5, and Q_2 gets tuples from S_5 and S_7, we will only consider source sets $\{S_1\}$,$\{S_5\}$,$\{S_1, S_5\}$,$\{S_7\}$, and $\{S_5, S_7\}$. We will not consider $\{S_1, S_7\}$, $\{S_1, S_5, S_7\}$, $\{S_2\}$, and many other source sets without any answer for either of the queries.

3.4.1. Generating AV Hierarchies. For now we will assume that all attributes have a discrete set of values, and we will also assume that the corresponding coverage and overlap statistics are available (see the last two paragraphs in this subsection regarding some important practical considerations). We now introduce GAVH (Generating AV Hierarchy, see Figure 14), an agglomerative hierarchical clustering algorithm ([HK00]), to automatically generate an AV Hierarchy for an attribute.

```
Algorithm GAVH()
    for (each attribute value)
        generate a cluster node $C$;
        feature vector $C.fv = (\overrightarrow{P(\widehat{S}|Q)}, P(Q))$;
        children $C.children = null$;
        put cluster node $C$ into AVQueue;
    end for
    while (AVQueue has more than two clusters)
        find the most similar pair of clusters $C_1$ and $C_2$;
        /* $d(C_1, C_2)$ is the minimum of all $d(C_i, C_j)$ */
        generate a new cluster $C$;
        $C.fv = (\frac{P(C_1) \times \overrightarrow{P(\widehat{S}|C_1)} + P(C_2) \times \overrightarrow{P(\widehat{S}|C_2)}}{P(C_1)+P(C_2))}, P(C_1) + P(C_2))$;
        $C.children = (C_1, C_2)$;
        put cluster $C$ into AVQueue;
        remove cluster $C_1$ and $C_2$ from AVQueue;
    end while
End GAVH;
```

Figure 8. The GAVH algorithm

The GAVH algorithm will build an AV Hierarchy tree, where each node in the tree has a feature vector summarizing the information that we maintain about an attribute value cluster. The feature vector is defined as: $(\overrightarrow{P(\widehat{S}|C)}, P(C))$, where $\overrightarrow{P(\widehat{S}|C)}$ is the coverage and overlap statistics vector of the cluster C for all the source sets and $P(C)$ is the class probability of the cluster C. Feature vectors are only used during the construction of AV hierarchies and can be removed afterwards. As we can see from Figure 14, we can incrementally compute a new cluster's coverage and overlap statistics vector $\overrightarrow{P(\widehat{S}|C)}$ by using the feature vectors of its children clusters C_1, C_2:

$$\overrightarrow{P(\widehat{S}|C)} = \frac{P(C_1) \times \overrightarrow{P(\widehat{S}|C_1)} + P(C_2) \times \overrightarrow{P(\widehat{S}|C_2)}}{P(C_1) + P(C_2))}$$

$$P(C) = P(C_1) + P(C_2)$$

Attribute Value Pre-Processing: The attribute values for generating AV hierarchies are extracted from the query list maintained by the mediator. Since the GAVH algorithm assumes that all attributes have discrete domains, we may need to preprocess the values of some types of attributes. For continuous numerical attributes, we divide the domain of the attribute into small ranges. Each range is treated as a discrete attribute value. For keyword-based attributes such as the attribute "title" in *BibFinder* we learn the frequently asked keyword sets using an item set mining algorithm. Each frequent keyword set will be treated as a discrete attribute value. Keyword sets that are rarely asked will not be remembered as attribute values.

Handling Low Selectivity Attribute Values: If an attribute value (i.e. a selection query binding value) is too general, some sources may only return a subset of answers to the mediator, while others may not even answer such general queries. In such cases the mediator will not be able to accurately figure out the number of tuples in these sources, and thus cannot know the coverage and overlap statistics of these queries to generate AV hierarchies. To handle this we use the coverage statistics of more specific queries in the query list to estimate the source coverage and overlap of the original queries. Specifically, we treat the original general queries as query classes, and to estimate the coverage of the sources for these general queries we use the statistics of the specific queries[7] within these classes using the following formula:

$$P(\widehat{S}|C) \doteq \frac{\sum_{Q \in C \; and \; (Q \; is \; specific)} P(\widehat{S}|Q)P(Q)}{\sum_{Q \in C \; and \; (Q \; is \; specific)} P(Q)}$$

As we can see, there is a slight difference between this formula and the formula for the definition of the overlap (or coverage) w.r.t. to class C. The difference is that here we only consider the overlap (or coverage) of specific queries within the class.

3.4.2. Flattening AV Hierarchies. Since the nodes of the AV Hierarchies generated using our GAVH algorithm contain only two children each, we may get a hierarchy with a large number of layers. One potential problem with such kinds of AV Hierarchies is that the level of abstraction may not actually increase when we go up the hierarchy. For example, in Figure 9, assuming the three attribute values have the same coverage and overlap statistics, then we should not put them into separate clusters. If we put these attribute values into two clusters C_1 and C_2, these two clusters are essentially in the same level of abstraction. Therefore we may waste our memory space on remembering the same statistics multiple times.

In order to prune these unnecessary clusters, we use another algorithm called FAVH (<u>F</u>lattening <u>AV</u> <u>H</u>ierarchy, see Figure 10). FAVH starts the flattening procedure from the root of the AV Hierarchy, then recursively checks and flattens the entire hierarchy.

To determine whether a cluster C_{child} should be preserved in the hierarchy, we compute the *tightness* of the cluster, which measures the accuracy of its statistics. We consider a cluster is tight if all the queries subsumed by the cluster (especially frequently asked ones)

[7]A query in the query list is called a specific query, if the number of answer tuples of the query returned by each source is less than the source's limitation.

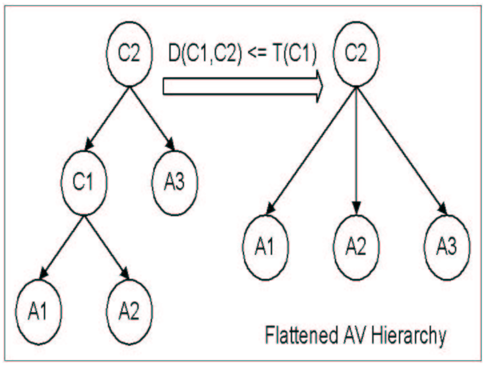


Figure 9. An example of flattening AV hierarchy

```
**Algorithm** *FAVH(clusterNode C)* //Starting from root;
  **if**($C$ has children)
    **for** (each child node $C_{child}$ in $C$)
      put $C_{child}$ into $Children\_Queue$
    **for (each node** $C_{child}$ **in** $Children\_Queue$**)**
      **if** ($d(C_{child}, C) <= \frac{1}{t(C_{child})}$)
        put $(C_{child}).children$ into $Children\_Queue$;
        remove $C_{child}$ from $Children\_Queue$;
      **end if**
    **for** (each children node $C_{child}$ in $Children\_Queue$)
      FAVH($C_{child}$);
  **end if**
**End** *FAVH*;
```

Figure 10. The FAVH algorithm

are close to its center. The *tightness* $t(C)$, of a cluster C, is calculated as following:

$$t(C) = \frac{1}{\sum_{Q \in C} \frac{P(Q)}{P(C)} d(Q, C)}$$

where $d(Q, C)$ is the distance between the query Q and the center of the cluster.

If the distance, $d(C_{child}, C)$, between a cluster and its parent cluster C is not larger than $\frac{1}{t(C_{child})}$, then we consider the cluster as unnecessary and put all of its children directly into its parent cluster.

3.5. Discovering Frequent Query Classes

As we discussed earlier, it may be prohibitively expensive to learn and keep in memory the coverage and overlap statistics for every possible query class. In order to keep the amount of statistics low, we would like to prune query classes which are rarely accessed. In this section we describe how frequently accessed classes are discovered in a two-stage process.

We use the term *candidate frequent class* to denote any class with class probability greater than the minimum frequency threshold *minfreq*. The example classes shown in Figure 6 with solid frame lines are candidate frequent classes. As we can see, some queries may have multiple lowest level ancestor classes which are candidate frequent classes and are not subsumed by each other. For example, the query (or class) (ICDE,01) has both the class (DB,01) and class (ICDE,RT) as its parent class. For a query with multiple ancestor classes, we need to map the query into a set of least-general ancestor classes which are not subsumed by each other (see Section 3.7). We will combine the statistics of these mapped classes to estimate the statistics for the query.

We also define the *class access probability* of a class C, denoted by $P_{map}(C)$, to be the probability that a random query posed to the mediator is actually mapped to the class C. It is estimated using the following formula:

$$P_{map}(C) = \sum_{Q \text{ is mapped to } C} P(Q)$$

Since the class access probability of a candidate frequent class will be affected by the distribution of other candidate frequent classes, in order to identify the classes with high class access probability, we have to discover all the candidate frequent classes first. In the next subsection, we will introduce an algorithm to discover candidate frequent classes. In

```
**Algorithm** *DFC(QList; minfreq : minimum support; n : # of classificatory attributes)*
  $classSet = \{\};$
  **for** $(k = 1; k <= n; k++)$
    Let $classSet_k = \{\};$
    **for** (each query $Q \in QList$)
      $C_Q = genClassSet(k, Q, ...);$
      **for** (each class $c \in C_Q$)
        **if** ($c \notin classSet_k$)   $classSet_k = classSet_k \cup \{c\};$
        $c.frequency = c.frequency + Q.frequency;$
      **end for**
    **end for**
    $classSet_k = \{c \in classSet_k | c.frequency >= minfreq\};$
    $classSet = classSet \cup classSet_k;$
  **end for**
  **return** $classSet;$
**End** *DFC*;
```

Figure 11. The DFC algorithm

Section 3.5.2, we will then discuss how to prune candidate frequent classes with low class access probability.

3.5.1. Discovering Candidate Frequent Classes. We present an algorithm, DFC (Discovering Candidate Frequent Classes, see Figure 1), to efficiently discover all the candidate frequent classes. The DFC algorithm dynamically prunes classes during counting and uses the *anti-monotone property*[8] ([HK00]) to avoid generating classes which are supersets of the pruned classes.

Specifically the algorithm makes multiple passes over the query list *QList*. It first finds all the candidate frequent classes with just one feature, then it finds all the candidate frequent classes with two features using the previous results and the anti-monotone property to efficiently prune classes before it starts counting, and so on. The algorithm continues until it gets all the candidate frequent classes with all the n features (where n is the total

[8]If a set cannot pass a test, all of its supersets will fail that test as well.

```
Procedure genClassSet(k : number of features;Q : the query; classSet : discovered frequent class set; AV hierarchies)
    for (each feature $f_i \in Q$)
        $ftSet_i = \{f_i\}$;
        $ftSet_i = ftSet_i \cup (\{ancestor(f_i)\} - \{root\})$;
    end for
    candidateSet={};
    for (each k feature combination $(ftSet_{j_1}, ..., ftSet_{j_k})$)
        $tempSet = ftSet_{j_1}$;
        for $(i = 1; i < k; i++)$
            remove any class $C \notin classSet_i$ from tempSet;
            $tempSet = tempSet \times ftSet_{j_{i+1}}$;
        end for
        remove any class $C \notin classSet_{k-1}$ from tempSet;
        $candidateSet = candidateSet \cup tempSet$;
    end for
    return candidateSet;
End genClassSet;
```

Figure 12. Ancestor class set generation procedure

number of classificatory attributes for which AV-hierarchies have been learned). For each query Q in the k-th pass, the algorithm finds the set of k feature classes the query falls in, and for each class C in the set, it increases the class probability $P(C)$ by the query probability $P(Q)$. The algorithm prunes the classes with class probability less than the minimum threshold probability $minfreq$.

The DFC algorithm finds all the candidate ancestor classes with k features for a query $Q = \{A_{c_1}, ..., A_{c_n}, frequency\}$ by procedure **genClassSet** (see Figure 2), where A_{c_i} is the feature value of the i^{th} classificatory attribute. The procedure prunes infrequent classes using the frequent class set *classSet* found in the previous $(k-1)$ passes. In order to improve the efficiency of the algorithm, it dynamically prunes infrequent classes during the cartesian product procedure.

Example: Assume we have a query Q={ICDE, 2001, 50} (here 50 is the query frequency) and $k = 2$. We first extract the feature(binding) values $\{A_{c_1} = ICDE, A_{c_2} = 2001\}$ from the query. Then for each feature, we generate a feature set which includes all the ancestors of the feature (see the corresponding AV Hierarchies in Figure 6) . This leads to two feature sets: $ftSet_1 = \{ICDE, DB\}$ and $ftSet_2 = \{2001\}$. Suppose the class with the single feature "ICDE" is not a frequent class in the previous results, then any class with the feature "ICDE" can not be a frequent class according to the anti-monotone property. We can prune the feature "ICDE" from $ftSet_1$, then we get the candidate 2-feature class set for the query Q,

$candidateSet = ftSet_1 \times ftSet_2 = \{DB\&2001\}$.

3.5.2. Pruning Low Access Probability Classes. The DFC algorithm will discover all the candidate frequent classes, which unfortunately may include many infrequently mapped classes. Here we introduce another algorithm, PLC (Pruning Low Access Probability Classes, see Figure 13), to assign class access probability and delete the classes with low access probability. The algorithm will scan the query list once, and map each query into a set of least-general candidate frequent ancestor classes (see Section 3.7). It then computes the class access probability for each class by counting the total frequencies of all the queries mapped to the class. The class with the lowest class access probability (less than $minfreq$) will be pruned, and the queries of the pruned classes will be re-mapped to other existing ancestor classes. The pruning process will continue until there is no class with access probability less than the threshold $minfreq$.

```
Procedure PLC(QList; classSet: frequent classes from DFC; minfreq)
    for (each $C \in classSet$)
        initialize $FR = 0$, and $FR_C = 0$ ;
    for(each query $Q$)
        Map $Q$ into a set of least-general classes $mSet$;
        for(each $C \in mSet$)
            $FR_C \leftarrow FR_C + FR_Q$;
            $FR = FR + FR_Q$;
        end for
    end for
    for(each class $C$)
        class access probability $P_{map}(C) \leftarrow \frac{FR_C}{FR}$;
    while (($\exists C \in classSet$) $P_{map}(C) < minfreq$)
        Delete the class with the smallest class access probability, $C'$, from $classSet$;
        Re-map the queries which are mapped to $C'$;
       for(new mapped class $C_{newMapped}$)
            recompute $P_{map}(C_{newMapped})$;
    end while
End PLC;
```

Figure 13. The PLC procedure

3.6. Mining Coverage and Overlap Statistics

For each frequent query class in the mediator, we learn coverage and overlap statistics. We use a minimum support threshold *minoverlap* to prune overlap statistics for uncorrelated source sets.

A simple way of learning the coverage and overlap statistics is to make a single pass over the $QList$, map each query into its ancestor frequent classes (see Section 3.7), and update the corresponding statistics vectors $\overrightarrow{P(\widehat{S}|C)}$ of its ancestor classes using the query's coverage and overlap statistics vector $\overrightarrow{P(\widehat{S}|Q)}$ through the formula $\overrightarrow{P(\widehat{S}|C)} = \frac{\sum_{Q \in C} \overrightarrow{P(\widehat{S}|Q)} \times P(Q)}{P(C)}$. When the mapping and updating procedure is completed, we simply need to prune the overlap statistics which are smaller than the threshold *minoverlap*. One potential problem of this naive approach is the possibility of running out of memory, since the system has to remember the coverage and overlap statistics for each source set and class combination. If the mediator has access to n sources and has discovered m frequent classes, then the memory requirement for learning these statistics is $m \times 2^n \times k$, where k is the number of bytes needed to store a float number. If $k = 1$, $m = 10000$, and the total number of memory available is $1GB$, this approach would not scale well when the number of sources is greater than 16.

In order to handle scenarios with large number of sources, we use a modified Apriori algorithm [AS94] to avoid considering any supersets of an uncorrelated source set. We first identify individual sources with coverage statistics greater than *minoverlap*, and keep coverage statistics for these sources. Then we discover all 2-*sourceSet* [9] with overlap greater than *minoverlap*, and keep only overlap statistics for these source sets. This process continues until we have the overlap statistics for all the correlated source sets.

3.7. Using Learned Coverage and Overlap Statistics

With the learned statistics, the mediator is able to find relevant sources for answering an incoming query. In order to access the learned statistics efficiently, both the learned AV hierarchies and the statistics for frequent query classes are loaded into hash tables in the mediator's main memory. In this section, we discuss how to use the learned statistics to estimate the coverage and overlap statistics for a new query, and how these statistics are used to generate query plans.

3.7.1. Query Mapping. In this section we describe the approach we use to map a user query into a set of frequent query classes. Given a new query Q, we first get all the

[9] *k-sourceSet* denotes the source sets with only k sources.

abstract values from the AV hierarchies corresponding to the binding values in Q. Both the binding values and the abstract values are used to map the query into query classes with statistics. For each attribute A_i with bindings, we generate a feature set $ftSet_{A_i}$ which includes the corresponding binding value and abstract values for the attribute. The mapped classes will be a subset of the candidate class set $cSet$:

$$cSet = ftSet_{A_1} \times ftSet_{A_2} \times ... \times ftSet_{A_n}$$

where n is the number of attributes with bindings in the query. Let $sSet$ denote all the frequent classes which have learned statistics and $mSet$ denote all the mapped classes of query Q. Then the set of mapped classes is:

$$mSet = cSet - \{C|(C \in cSet) \cap (C \notin sSet)\} - \{C|(\exists C' \in (sSet \cap cSet))(C' \subset C)\}$$

In other words, to obtain the mapped class set we remove all the classes which do not have any learned statistics as well as the classes which subsume any class with statistics from the candidate class set. The reason for the latter is because the statistics of the subsumed class are more specific to the query.

Once we have the relevant class set, we compute the estimated coverage and overlap statistics vector $\overrightarrow{P(\widehat{S}|Q)}$ for the new query Q using the statistics vectors of the mapped classes $\overrightarrow{P(\widehat{S}|C_i)}$ and their corresponding tightness information $t(C_i)$.

$$\overrightarrow{P(\widehat{S}|Q)} = \sum_{C_i} \frac{t(C_i)}{\sum t(C_i)} \overrightarrow{P(\widehat{S}|C_i)}$$

Since the classes with large tightness values are more likely to provide more accurate statistics, we give more weight to query classes with large tightness values.

Using Coverage and Overlap Statistics to Generate Query Plans:

3.7.2. Computing Residual Coverage. Once we have the coverage and overlap statistics, we use the **Simple Greedy** and **Greedy Select** algorithms described in [FKL97] to generate query plans. Specifically, *Simple Greedy* generates plans by greedily selecting the top k sources ranked only according to their coverage, while *Greedy Select* selects sources with high residual coverage calculated using both the coverage and overlap statistics. The top k source selection using both coverage and overlap statistics is essentially a maximum coverage problem [H97]. It is well known that a simple greedy algorithm (such as the greedy select algorithm [FKL97]) solves the maximum coverage problem approximately within a factor of $(1 - e^{-1})$ of optimum [CFN77]. Feige [F96] proved that no polynomial algorithm can have better worst-case performance guarantee.

In this section we discuss how we compute the residual coverages. In order to find a plan with top k sources, we start by selecting the source with the highest coverage [FKL97] as the first source. We then we use the overlap statistics to compute the residual coverages of the rest of the sources to find the second best, given the first; the third best, given the first and second, and so on, until we get a plan with the desired coverage.

In particular, after selecting the first and second best sources S_1 and S_2 for the class C, the residual coverage of a third source S_3 can be computed as:

$$P(S_3 \wedge \neg S_1 \wedge \neg S_2 | C) = P(S_3|C) - P(S_3 \wedge S_1|C) - P(S_3 \wedge S_2|C) + P(S_3 \wedge S_2 \wedge S_1|C)$$

where, $P(S_i \wedge \neg S_j)$ is the probability that a random tuple belongs to S_i but not to S_j. In the general case, after we had already selected the best n sources $\widehat{S} = \{S_1, S_2, ..., S_n\}$, the residual coverage of an additional source S can be expressed as:

$$P(S \wedge \neg \widehat{S}|C) = P(S|C) + \sum_{k=1}^{n}[(-1)^k \sum_{\widehat{S}^k \subseteq \widehat{S} \wedge |\widehat{S}^k| = k} P(S \wedge \widehat{S}^k|C)]$$

where $P(S \wedge \neg \widehat{S}|C)$ is shorthand for $P(S \wedge \neg S_1 \wedge \neg S_2 \wedge ... \wedge \neg S_n|C)$.

A naive evaluation of this formula would require 2^n accesses to the database of learned statistics, corresponding to the overlap of each possible subset of the n sources with source S. It is however possible to make this computation more efficient by exploiting the structure of the stored statistics. Specifically, recall that we only keep overlap statistics for source sets with sufficient number of overlap tuples, and assume that source sets without overlap statistics are disjoint (thus their probability of overlap is zero). Furthermore, if the overlap is zero for a source set $\widehat{S}$, we can ignore looking up the overlap statistics for supersets of $\widehat{S}$, since they will all be zero by the anti-monotone property.

To illustrate the above, suppose S_1,S_2,S_3 andS_4 are sources exporting tuples for class C. Let $P(S_1|C)$, $P(S_2|C)$ $P(S_3|C)$ and $P(S_4|C)$ be the learned coverage statistics, and $P(S_1 \wedge S_2|C)$ and $P(S_2 \wedge S_3|C)$ be the learned overlap statistics. The expression for computing the residual coverage of S_3 given that S_1 and S_2 are already selected is:

$$P(S_3 \wedge \neg S_1 \wedge \neg S_2|C) = P(S_3|C) - \underbrace{P(S_3 \wedge S_1|C)}_{=0} - P(S_3 \wedge S_2|C) + \underbrace{P(S_3 \wedge S_1 \wedge S_2|C)}_{=0 \text{ } since \text{ } \{S_3,S_1\} \subseteq \{S_2,S_1,S_2\}}$$

We note that once we know $P(S_3 \wedge S_1|C)$ is zero, we can avoid looking up $P(S_3 \wedge S_1 \wedge S_2|C)$, since the latter set is a superset of the former.

In Figure 14, we present an algorithm that uses this structure to evaluate the residual coverage in an efficient fashion. In particular, this algorithm will cut the number of statistics lookups from 2^n to $\mathcal{R}+n$, where $\mathcal{R}$ is the total number of overlap statistics remembered for class C and n is the total number of sources already selected. This consequent efficiency is critical in practice since computation of residual coverage forms the inner loop of any query processing algorithm that considers source coverage.

The inputs to the algorithm in Figure 14 are the source s for which we are going to compute the residual coverage, and the currently selected set of sources $\widehat{S}_s$. The auxiliary

```
Algorithm residualCoverage (s: source; $\widehat{S}_s$: selected sources;
$\widehat{S}_c$: constraint source set)
  $n$ = the number of sources in $\widehat{S}_s$;
  if ($\widehat{S}_c \neq \emptyset$)    then $p$ = the position of $\widehat{S}_c$'s last source in $\widehat{S}_s$;
  else $p$=0;
  Let resCoverage = 0;
  if the overlap statistics for the source set $\widehat{S}_c \cup \{s\}$
   are present in the learned statistics;
      //This means their overlap is > minoverlap.
     for ($i = p + 1$; $i \leq n$; $i++$)
       Let $\widehat{S}'_c = \widehat{S}_c \cup \{the\ i^{th}$ source in $\widehat{S}_s\}$;
       //keep order of sources in $\widehat{S}'_c$ same as in $\widehat{S}_s$
       resCoverage = resCoverage+residualCoverage(s, $\widehat{S}_s$, $\widehat{S}'_c$);
     end for
     resCoverage = resCoverage + $(-1)^{|\widehat{S}_c|}$overlap;
  end if
  return resCoverage;
End residualCoverage;
```

Figure 14. Algorithm for computing residual coverage

datastructure $\widehat{S}_c$, initially set to $\emptyset$, is used to restrict the source overlaps considered by the *residualCoverage* algorithm. In each invocation, the algorithm first looks for the overlap statistics for $\{s\} \cup \widehat{S}_c$. If this statistic is among the learned (stored) statistics, the algorithm recursively invokes itself on supersets of $\{s\} \cup \widehat{S}_c$. Otherwise, the recursion stops in that branch (eliminating all the redundant superset lookups).

3.8. Experimental Setup

The statistics learning system *StatMiner* has been fully implemented and evaluated using both controlled datasets and *BibFinder*, a popular computer science bibliography mediator that we developed. In the *BibFinder* experiments, we show the effectiveness of our approach in a real scenario, where we can not control the data distribution over these online Web sources. we evaluated both the frequency-based approach and the size-based approach in *BibFinder*. In the experiment with controlled datasets, we focused on evaluating the efficiency and effectiveness of our size-based approach for scenarios with large number of sources. We will start by describing the experimental setup for both *BibFinder* and controlled dataset scenarios.

3.8.1. *BibFinder* Testbed.

We use *BibFinder* as a testbed to evaluate our ability to learn an approximate data distribution from real Web data. Both the frequency-based approach and the size-based approach have been evaluated in *BibFinder*. In order to evaluate the frequency-based approach, we use the real user queries submitted to *BibFinder* to observe whether it will work well with real query distributions. In order to evaluate the effectiveness of the size-based approach over *BibFinder*, we probe the online sources integrated by *BibFinder* to get a initial query list. The focus is to test the effect of various probing techniques we use. we will explain the detailed experimental setup for both approaches below.

3.8.1.1. *Experimental Setup for Evaluating the Frequency-based Approach.*

Five structured Web bibliography data sources in *BibFinder* are used in our experimental evaluation: DBLP, CSB, ACM DL, Science Direct and Network Bibliography. We used the 25000 real queries asked by *BibFinder* users as of May 20, 2003 as the query list. Among them, we randomly chose 4500 queries as test queries and the others were used as training data. The AV Hierarchies for all four attributes were learned automatically using our GAVH algorithm. The learned Author hierarchy has more than 8000 distinct values,[10] the Title hierarchy keeps only 1200 frequently asked keyword itemsets, the Conference hierarchy has more than 600 distinct values, and the Year hierarchy has 95 distinct values. Note that

[10]Since it is too large for GAVH to learn upon it directly, we first group these 8000 values into 2300 value clusters using a radius based clustering algorithm ($O(n)$ complexity), and use GAVH to generate a hierarchy for these 2300 value clusters.

we consider a range query (for example: ">1990") as a single distinct value. In Figure 15, we show a learned attribute value hierarchy for the conference attribute. As we can see, the hierarchy starts with the HRoot node, followed by the highest level intermediate nodes, such as qc140, qc396, ..., qc397 and some conferences: "information,processing,letters"[11], "issta", and, "future,generation,computer,systems". Since the coverage and overlap statistics of these conferences are very different from those of other conferences, they are put directly under the root node of the hierarchy. Note that the names of intermediate nodes are generated automatically by the GAVH algorithm. Naturally the conferences within an intermediate cluster nodes are only guaranteed to be similar in terms of their coverage and overlap statistics and may not be similar in terms of their semantic meaning at all. For example, the "ieee,computer" journal and the "icml" conference are put into the same cluster qc350 because they are similar in terms of their coverage and overlap statistics, however they focus on very different topics.

[11]Note that commas are used in our implementation to separate the keywords within a conference name.

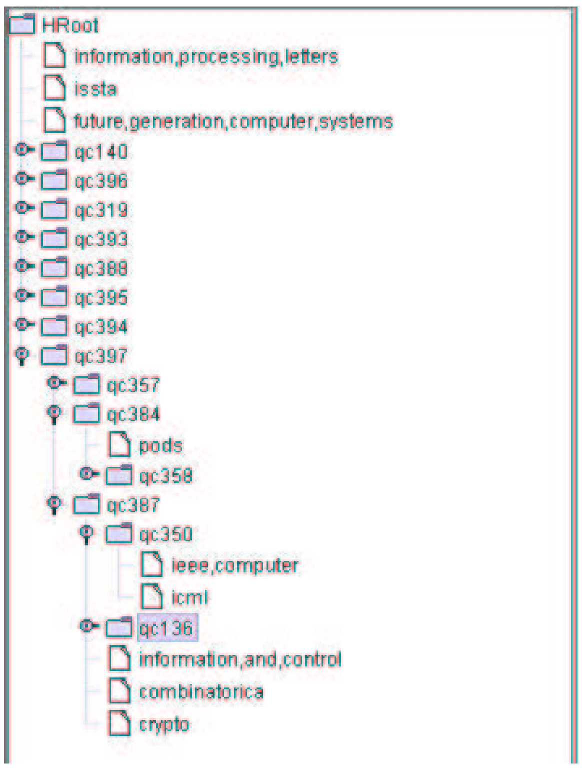


Figure 15. Learned attribute value hierarchy for the conference attribute. Note that only the last cluster node (i.e. qc397) of the highest level nodes has been unfolded.

3.8.1.2. *Experimental Setup for Evaluating the Size-based Approach.* Six structured Web bibliography data sources in *BibFinder* : DBLP, CSB, ACM DL, ACM Guide, Science Direct and IEEExplore are used in our experimental evaluation. We chose **paper(title, author, conference/journal, year)** as the mediated relation. conference/journal and year are chosen as the classificatory attributes. Since it's difficult to get a good AV hierarchy for the conference/journal attribute, we use the GAVH and FAVH described in Section 3.4 to automatically learn the conference/journal hierarchy. We gathered 604 conference and journal names from DBLP, ACM dl, and Science Direct Web pages. These names are used to generate probing queries and to generate AV hierarchy for the conference/journal attribute. The AV hierarchy for the year attribute is consisted of the years from 1954 to 2003 as leaf nodes. Every five years in first level of the hierarchy is subsumed by a second level ancestor node, and every ten years are subsumed by a third level ancestor node, and ROOT is the only node in the fourth level. The space of all the probing queries is the cartisian product of the 604 conference/journal names and the 50 years. We used a set of 578 real queries asked by *BibFinder* users as the test queries.

Probing Data Sources: To evaluate our size-based approach in *BibFinder*, we must probe its mediated sources to estimate the spread of data over them. As discussed earlier, we generate the sample queries by taking cartesian products of the conference/journal names and recent 50 years. At this time we are assuming that only queries binding both conference/journal and year will be considered "safe" by the Web sources.

Probing Web sources using all the queries we can generate will be too costly for large number of queries. Hence we use query sampling to select a smaller set of queries to generate the required data distribution of sources.

Query Sampling: As mentioned in Section 3.3.2, we generate the set of sample probing queries using both *Simple Random Sampling* and *Stratified Random Sampling*. After generating the set of spanning queries we use the two sampling approaches to extract a sample set of queries to probe the data sources. Simple Random sampling picks the samples from the complete set of queries, whereas to employ the Stratified Random sampling approach, we have to further classify the queries into various *strata*. The strata is chosen as the abstract feature of any one classificatory attribute say $A1$. All the queries that bind $A1$ using leaf values subsumed by a strata are mapped to that strata. A strata based on an abstract feature that only subsumes leaf nodes will have fewer queries mapped to it compared to the strata that is based on an abstract feature that subsumes both the leaf nodes and other abstract features. Thus the level of abstraction at which we decide a strata varies the number of queries that get mapped to the strata. The lowest abstraction is the leaf node, while the root gives highest abstraction. Selecting root as the strata will make Stratified Random Sampling equal to Simple Random, where selecting the leaf nodes as strata, will be equal to issuing all the spanning queries.

3.8.2. Controlled Datasets for Evaluating the Size-based Approach. To evaluate our size-based approach over controlled data sources, we set up a set of "remote" data sources accessible on the Internet. The sources were populated with synthetic data generated using the data generator from TPC-W benchmark [TPC] (see below). The TPC sources support controlled experimentation as their data distribution (and consequently the coverage and overlap among web sources) can be varied by us.

We designed 25 sources using 200000 tuples for the relation Books. We chose **Books**(Bookid, Pubyear, Subject, Publisher, Cover) as the relation exported by our sources. The decision to use Books as the sample schema was motivated by the fact that multiple autonomous Internet sources projecting this relation exist, and in the absence of statistics about these sources, only naive mediation services are currently provided. Pubyear, Subject and Cover are used as the *classificatory* attributes in the relation Books. The hierarchies were designed as shown in Figures 16 and 17. To evaluate the effect of the resolution of the hierarchy on ranking accuracy we designed two separate hierarchies for Subject, containing 180 and 40 leaf nodes respectively. Leaf node values for Pubyear range from 1980 to 2001, while Cover is relatively small with only five leaf nodes. The Subject hierarchy was modeled from the classification of books given by the online bookstore Amazon [AM]. We populated the data sources exporting the mediator relation using DataGen, the data generator from

TPC-W Benchmark [TPC]. The distribution of data in these sources was determined by controlling the values used to instantiate the classificatory attributes Pubyear, Subject and Cover. For example, two sources S_1 and S_2 both providing tuples under abstract feature "Databases" of Subject hierarchy, are designed to have varying overlap with source S_3, by selecting different subsets of features under "Databases" to instantiate the source tuples. These subsets may be mutually exclusive, but they overlap with the subset of features selected for populating source S_3. Since the actual generation of data sources is done by using DataGen, the above mentioned procedure gives us a macro level control over the design of overlap among sources. In fact DataGen populates the sources by initializing each attribute of the Books relation using a randomly chosen value from a list of seed values for that attribute. Hence we control the query classes for which the sources provide answer tuples and may overlap with other sources but not the actual values of coverage and overlap given by sources.

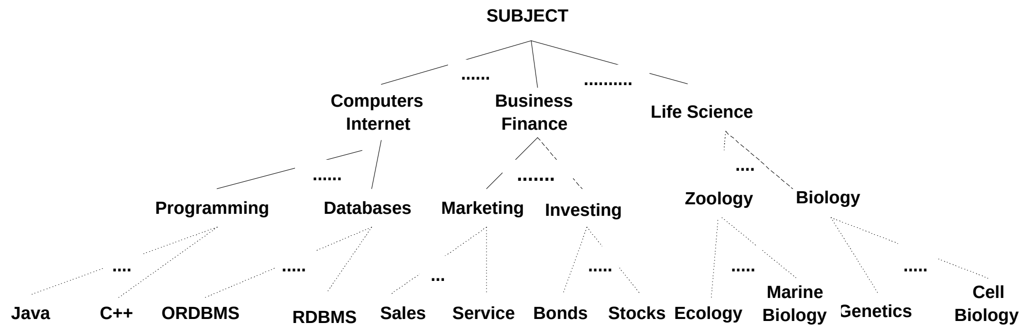


Figure 16. Subject hierarchy

3.8.3. Algorithms and Evaluation Metrics. To evaluate the accuracy of the statistics learnt by *StatMiner* we tested them using two simple plan generation algorithms. Our mediator implements the **Simple Greedy** and **Greedy Select** algorithms described in [FKL97] to generate query plans using the source coverage and overlap statistics learnt

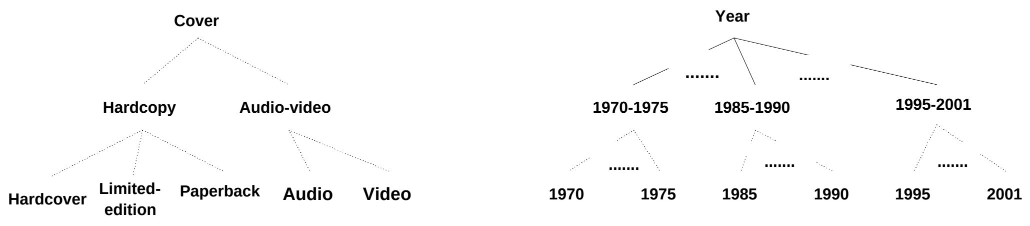


Figure 17. Cover and year hierarchy

by *StatMiner*. Given a query, Simple Greedy generates a plan by assuming all sources are independent and greedily selects top k sources ranked according to their coverages. On the other hand, Greedy Select generates query plans by selecting sources with high residual coverages calculated using both the coverage and overlap statistics (see Section 3.7.2).

We evaluate the plans generated by both the planners for various sets of statistics learnt by *StatMiner* for differing threshold values and AV hierarchies. We compare the precision of plans generated by both the algorithms. We define the *plan precision* to be the fraction of sources in the estimated plan, which turn out to be the real top k sources after we execute the query. Let $TopK$ refer to the real top k sources, and $Selected(p)$ refer to the k sources selected in the plan p. Then the *precision* of the plan p is:

$$precision(p) = \frac{|TopK \cap Selected(p)|}{|Selected(p)|}$$

The average precision and number of answers returned by executing the plan are used to estimate the accuracy of the learned statistics.

We also measure the *absolute error* between the estimated statistics and the real coverage and overlap values. The *absolute error* is computed using the following formula:

$$\frac{\sum_{Q \in TestQuerySet} \sqrt{\sum_i [P'(\widehat{S}_i|Q) - P(\widehat{S}_i|Q)]^2}}{|TestQuerySet|}$$

where $\widehat{S}_i$ denotes the i^{th} source set of all possible source sets in the mediator, $P'(\widehat{S}_i|Q)$ denotes the estimated overlap (or coverage) of the source set $\widehat{S}_i$ for query Q, $P(\widehat{S}_i|Q)$

denotes the real overlap (or coverage) of the source set $\widehat{S}_i$ for query Q, and $TestQuerySet$ refers to the set of all test queries.

3.9. Experimental Results

We now present the results of both experiments using *BibFinder* as testbed, and experiments using controlled datasets.

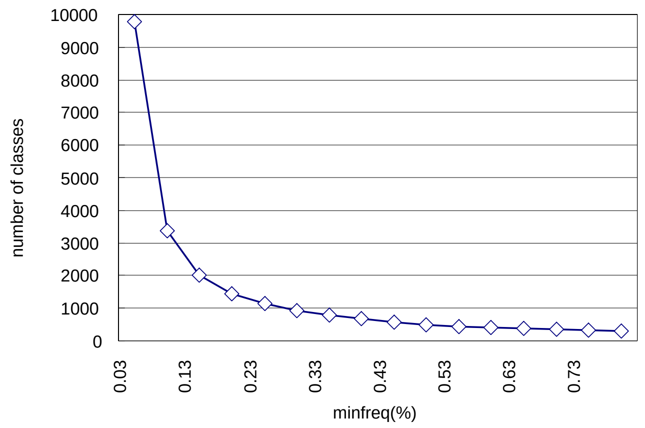


Figure 18. The total number of classes learned

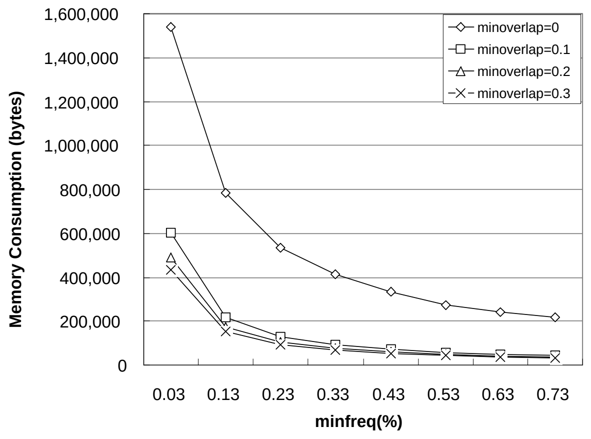


Figure 19. The total amount of memory needed for keeping the learned statistics in BibFinder

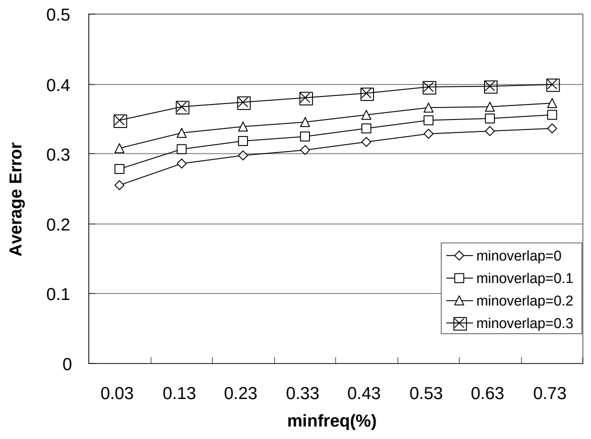


Figure 20. The average distance between the estimated statistics and the real coverage and overlap values.

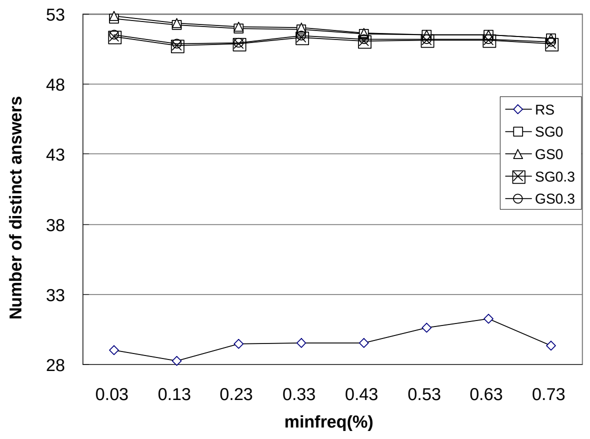


Figure 21. The average number of answers *BibFinder* returns by executing the query plans with top 2 sources.

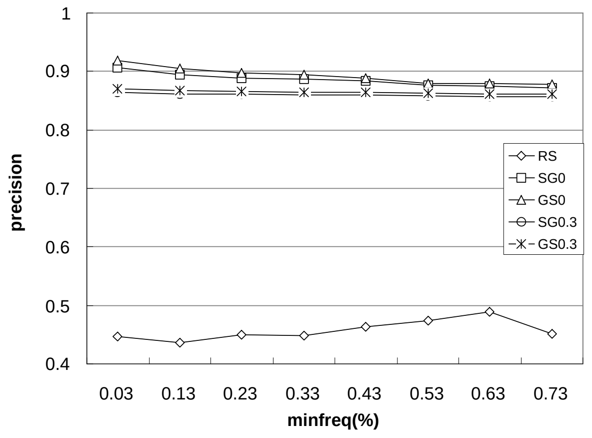


Figure 22. Precision for query plans with top 2 sources.

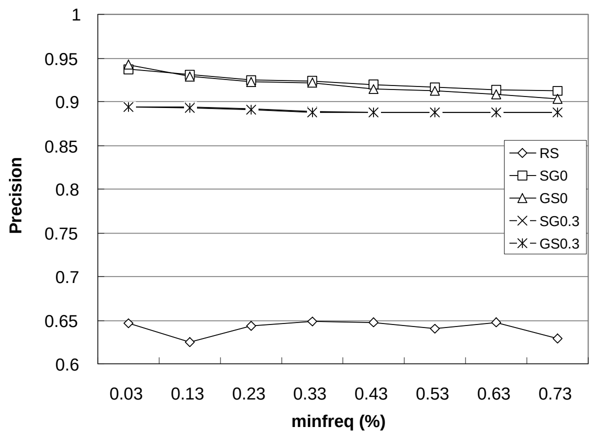


Figure 23. Precision for query plans with top 3 sources.

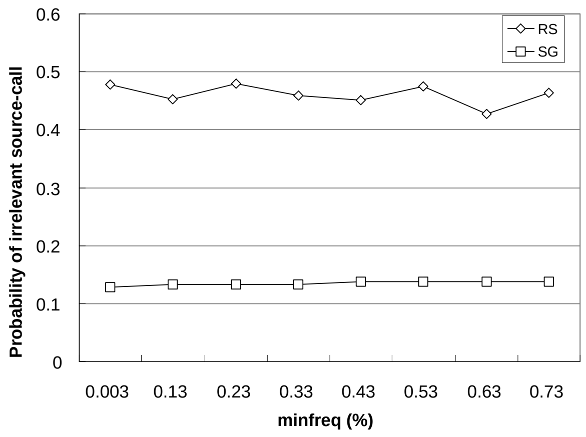


Figure 24. The percent of the total source-calls that are irrelevant for query plans with top 1 sources.

3.9.1. Results of Evaluating Frequency-based Approach over *BibFinder*.

Space Consumption for Different *minfreq* and *minoverlap* Thresholds: In Figures 18 and 19, we observe the reduction in space consumption and number of classes when we increase the *minfreq* and *minoverlap* thresholds. Slightly increasing the *minfreq* threshold from 0.03% to 0.13% causes the number of classes to drop dramatically from approximately 10000 classes to 3000. As we increase the *minfreq* threshold, the number of classes decreases, however the decrease rate becomes smaller as the threshold becomes larger. In Figure 19, we observe the size of the required memory for different levels of abstraction of the statistics. Clearly, as we increase any of these two thresholds the space consumption drops, however the pruning power also drops simultaneously.[12]

Accuracy of the Learned Statistics for Different *minfreq* and *minoverlap* Thresholds: Figure 20 plots the absolute error of the learned statistics for the 4500 test queries. The graph illustrates that although the error increases as any of these two thresholds increase, the increase rates remain almost the same. There is no dramatic increase after the initial increases of the thresholds. If we looked at both Figures 19 and 20 together, we can see that the absolute error of threshold combination: $minfreq = 0.13\%$ and $minoverlap = 0.1$ is almost the same as that of $minfreq = 0.33\%$ and $minoverlap = 0$, while the former uses only 50% of the memory required by the latter. This fact tells us that keeping very detailed overlap statistics of uncorrelated source sets for general query classes would not necessarily increase the accuracy of our statistics while requiring much more space.

Effectiveness of the Learned Statistics: We evaluate the effectiveness of the learned statistics by actually testing these statistics in *BibFinder* and observing the precision of the query plans and the number of distinct answers returned from the Web sources when we execute these plans to answer user queries.

Note that in all the figures described below, RS refers to Random Select algo-

[12]Note that for a better readability of our plots, we did not include the number of classes and memory consumption when the *minfreq* threshold is equal to zero, as the corresponding values were much larger than those obtained for other threshold combinations. In fact, the total number of classes when the *minfreq* is equal to zero is about 540000, and the memory requirement when both *minfreq* and *minoverlap* are equal to zero is about 40MB. Although in our current experiment setting 40MB is the maximal memory space needed to keep the statistics (mainly because *BibFinder* is at its beginning stage), the required memory could become much larger as the number of users and the number of integrated sources grow.

rithm, SG0 refers to Simple Greedy algorithm with *minoverlap* = 0, GS0 refers to Greedy Select algorithm with *minoverlap* = 0, SG0.3 refers to Simple Greedy algorithm with *minoverlap* = 0.3, and GS0.3 refers to Greedy Select algorithm with *minoverlap* = 0.3.

In Figure 21, we observe how the *minfreq* and *minoverlap* thresholds influence the average number of distinct answers returned by *BibFinder* for the 4500 test queries when executing query plans with top 2 sources. As indicated by the graph, for all the threshold combinations, we always get on average more than 50 distinct answers when using our learned statistics and query plans selected by Simple Greedy and Greedy Select, while we can only get about 30 distinct answers by randomly selecting 2 sources. In Figures 22 and 23, we observe the average precision of the top 2 and top 3 sources ranked using statistics with different level of abstraction for the test queries. As we can see, the plans using our learned statistics have high precision, and their precision decreases very slowly as we change the *minfreq* and *minoverlap* thresholds.

One fact we need to point out is that the precision of the plans using Simple Greedy and Greedy Select algorithm are very close (although Greedy Select is a little better most of the time). This is not as we expected, since the Simple Greedy only uses the coverage statistics, while Greedy Select uses both coverage and overlap statistics. When we studied many queries asked by the *BibFinder* users and the corresponding coverage and overlap statistics, we found that the distribution of answer tuples over sources integrated by *BibFinder* almost follow independence assumption for most of the queries asked by the users. However in other scenarios Greedy Select can perform considerably better than Simple Greedy. For instance, in our experiment with a controlled data set, where we set 25 artificial sources including some highly correlated sources, we did find that the plans generated by Greedy Select were significantly better than those generated by Simple Greedy.

Figure 24 shows the possibility of a source call being a completely irrelevant source call (i.e. the source has no answer for the query asked). The graph reveals that the most relevant source selected using our algorithm has only 12% possibility of being an irrelevant source call, while the randomly picked source has about 46% possibility. This illustrates that by using our statistics *BibFinder* can significantly reduce the unnecessary load on its integrated sources.

Efficiency Issues: We now discuss the time needed for learning and using the coverage and overlap statistics. All our experiments were run under JDK 1.2.2 on a 500MHZ SUN-Blade-100 with 768Mb of RAM. From the experiments, we found that using the learned statistics to generate query plans for a new query is very fast, specifically always less than 1 millisecond. In terms of the statistics learning, costs associated with discovering frequent query classes and learning statistics are also fairly inexpensive (i.e. always less than 100 seconds). Our experiments with 25 artificial sources (see 3.9.3) also shows that our statistics learning algorithms can scale well. The most expensive phase is learning the AV Hierarchies. During the experiments we found that the GAVH algorithm can be very time-consuming when the number of attribute values is large. Specifically, it takes us 719ms to learn the Year hierarchy, 1 minute to learn the Conference hierarchy, 25 minutes to learn the Title keywords hierarchy, and 2.5 hours to learn the Author hierarchy. However since GAVH runs offline and only needs to run once, it still is not a major drawback. Since it is the most time consuming phase, we can consider incrementally updating the hierarchy as new queries come in.

3.9.2. Results of Evaluating the Size-based Approach over *BibFinder*.

Given that the cost of probing tends to dominate the statistics gathering approach, we wanted to see how accurate the learned statistics are with respect to the two probing strategies. We used *BibFinder* sources for evaluating the probing strategies. The set of probing queries are generated by taking a cartesian product of the values of the conference/journal attribute and year attribute. The total number of queries generated is 30200. In order to be polite to the Web sources, we probe them at the rate of 3 queries per minute.

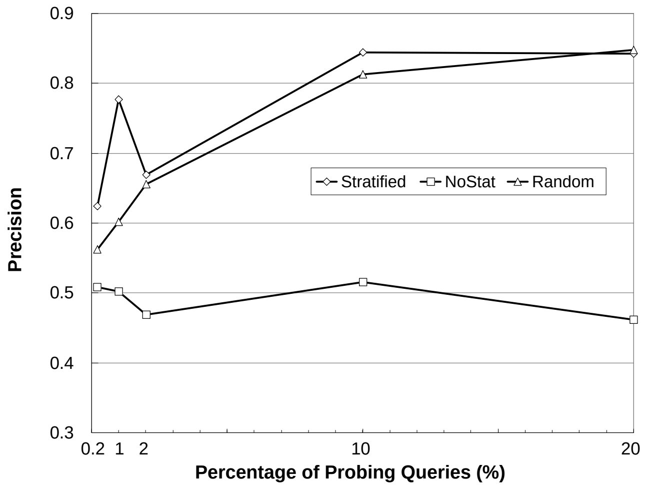


Figure 25. Comparing average precision of query plans for top 3 sources obtained using statistics through different probing strategies

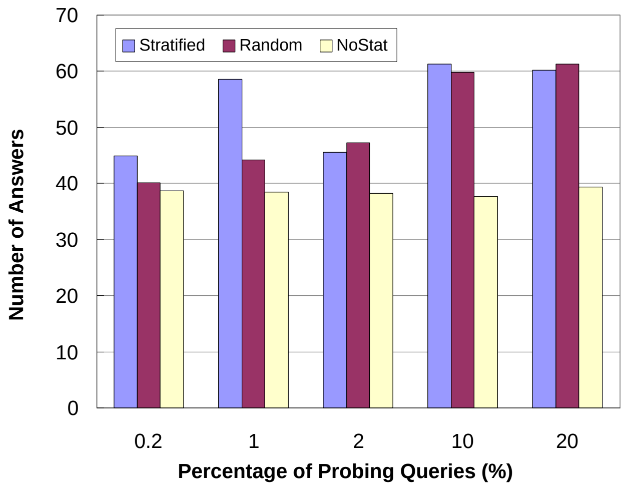


Figure 26. Comparing average number of answers returned by *BibFinder* by executing query plans for top 3 sources obtained using statistics through different probing strategies

In the experiment we study the efficiency of different probing strategies and different number of probing queries by observing the effectiveness of the learned statistics through their probing results.

For each of these two probing strategies, we generate five sets of probing queries with different number of probing queries: 60 (0.2%), 302(1%), 604(2%), 3020(10%), 6040(20%). Specifically in the stratified sampling, the first set of 60 probing queries is generated by randomly selecting 60 conference/journal names without replacement and for each name randomly selecting a year from the 50 years; the second set of 302 queries is generated by randomly selecting 60 conference/journal names and for each names selecting 5 years (one from year ten year period); the third set of 602 queries is generated by selecting all names and for each names randomly selecting a year; the fourth set of 3020 queries is generated by selecting all names and for each names selecting 5 years (one year from each 10 year period); the fifth set of 6040 queries is generated by selecting all names, and for each name selecting 10 years (one year from each 5 year period). In the random sampling strategies, we generate the sets of queries by randomly selecting 60, 302, 604, 3020, and 6040 queries from all the 30200 queries.

We now describe the steps in the experimental procedure:

1. For each of the probing strategies and for each set of probing queries, we probe all the six online sources;
2. Generate conference/journal hierarchies using the probing results;
3. Discover large query classes using the learned conference/journal hierarchy and the year hierarchy using the probing results;
4. Learn coverage and overlap statistics for each discovered large classes using the probing results
5. Use a list of 578 real user queries submitted to *BibFinder* to evaluate the learned statistics.

In Figure 25, we observe the average precision of query plans for top 3 sources for different sampling strategies and different number of probing queries. Here we fix the thresholds $minfreq = 0.1\%$ and $minoverlap = 1\%$. The query plans are generated by the greedy select algorithm using the learned coverage and overlap statistics. Different probing strategies and different number of probing queries affect the precision of the learned statistics, which affect the plan precision. From the figure, we can see that the stratified sampling is doing better than random sampling when the number of probing queries is small and the selection of strata is good, especially for the set of 302 probing queries. For each conference/journal, probing five years in each ten year period is much better than for each conference/journal randomly probing one year in a fifty year period. This is because the large classes discovered using 5 year probing results are more likely to be important conferences/journals than those using one year probing results. Learning the distribution over the sources for important conferences/journal will improve the precision, since users are more interested in these conferences/journals and the statistics for these conferences are more representative than that of random conferences/journals. However as the number of probing queries increases, the difference between random and stratified sampling becomes smaller (as to be expected).

In Figure 26, we observe the average number of answers from *BibFinder* when executing query plans for 3 sources for the 578 user queries. The figure illustrates results for different probing strategies and different number of probing queries. As we can see, the result is quite consistent with the plan precision. It is interesting to note that when using the statistics learned from the stratified probing results of 302 queries, *BibFinder* can actually provide about 50% more answers than by randomly querying 3 sources without using any statistics.

The above results are encouraging and show that our approach of leaning and using coverage and overlap statistics in *BibFinder* is able to give good results even for a very small sample of all probing queries. They also show that the stratified sampling is doing much better than random sampling when a good stratification strategy is chosen, and the number of probing queries is relatively small.

3.9.3. Results of Evaluating the Size-based Approach over Controlled Data Sources. Now we present results of experiments conducted to study the variation in pruning power, the efficiency and the accuracy of *StatMiner* in the controlled experiment.

In particular, given a set of sources and probing queries, our aim is to show that we can trade time and space for accuracy by increasing the threshold $minfreq$. Specifically by increasing threshold $minfreq$, the ***time*** (to identify large classes) and ***space*** (number of large classes remembered) usage can be reduced with a reduction in ***accuracy*** of the learnt estimates. All the experiments presented here were conducted on a 500MHZ Sun-Blade-100 systems with 256MB main memory running under Solaris 5.8. The sources in the mediator are hosted on a Sun Ultra 5 Web server located on campus.

Effect of Hierarchies on Space and Time: To evaluate the performance of our statistics learner, we varied $minfreq$ and measured the number of large classes and the time utilized for learning source coverage statistics for these large classes. Figure 27 compares the time taken by *StatMiner* to learn rules for different values of $minfreq$. Figure 28 compares the number of pruned classes with increase in value of $minfreq$. We represent $minfreq$ as a percentage of the total number of tuples in the relation. The total tuples in the relation is calculated as the number of unique tuples generated by the probing queries.

As can be seen from Figure 27, for lower values of threshold $minfreq$, *StatMiner* takes more time to learn the rules. For lower values of $minfreq$, *StatMiner* will prune less number of classes and hence for each class, *StatMiner*will generate large number of rules. This in turn explains the increase in learning time for lower threshold values.

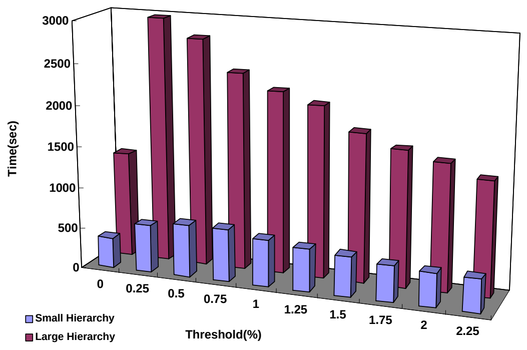


Figure 27. *StatMiner* learning time for various thresholds

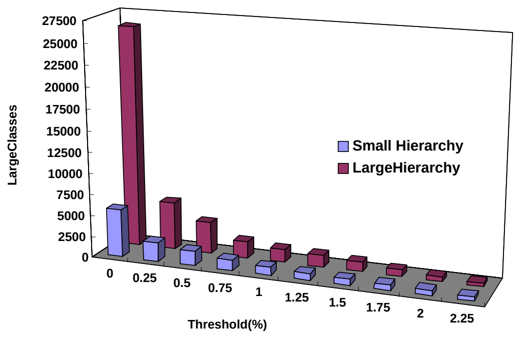


Figure 28. Pruning of classes by *StatMiner*

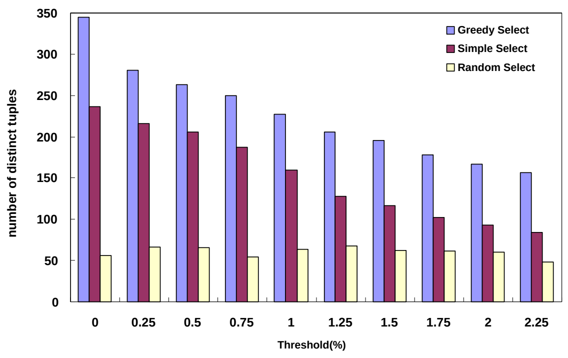


Figure 29. Comparing average number of answers by executing query plans for top 5 sources obtained by different planning algorithms using learned statistics

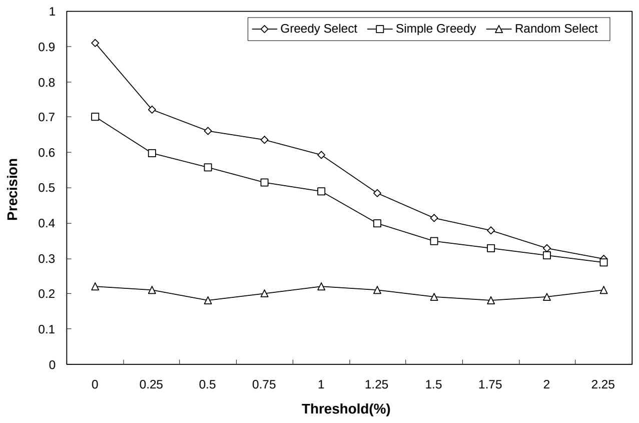


Figure 30. Comparing average precision of query plans for top 5 sources obtained by different planning algorithms using learned statistics

In Figure 28, with increase in value of $minfreq$, the number of small classes pruned increase and hence we see a reduction in the number of large classes. For any value of $minfreq$ greater than the support of the largest abstract class in the classSet, *StatMiner* returns only the root as the class to remember. Figures 27 and 28 show *StatMiner* performing uniformly for both Small and Large hierarchy. For both hierarchies, *StatMiner* generates large number of classes for small threshold values and requires more learning time. From Figures 27 and 28, we can see that the amount of time used and classes generated (space requirement) for the Large hierarchy is considerably higher than for Small hierarchy.

Accuracy of Estimated Coverages: To calculate the error in our coverage estimates, we used the prototype implementations of “Simple Greedy” and “Greedy Select” algorithms and a subset of our spanning queries as test queries. Since the test queries will have classificatory attributes bound, from Section 3.7 we see that the mediator maps them to the lowest abstract classes for which coverage statistics have been learnt. Once the query is mapped to a class, the mediator then generates plans using the ranking algorithms, Simple Greedy and Greedy Select as described in Section 3.8.3. We compare the plans generated by these algorithms with a naive plan generated by **Random Select**. Random select algorithm arbitrarily picks k sources without using any statistics. The source rankings generated by all the three algorithms is compared with the “true ranking” determined by querying all the sources. Figure 30 compares the precision of plans generated by the three approaches with respect to the true ranking of the sources.

As can be seen from Figure 29 for all values of $minfreq$ Greedy Select gives the best plan, while Simple Greedy is close second, but the Random Select performs poorly. The results are according to our expectations, since Greedy Select generates plans by calculating residual coverage of sources and thereby takes into account the amount of overlap among sources, while Simple Greedy calls sources with high coverages thereby ignoring the overlap statistics and hence generates less number of tuples.

In Figure 30 we compare the precision of plans generated by the three approaches. We define the precision of a plan to be the fraction of sources in the estimated plan, which turn out to be the real top k sources after we execute the query. Figure 30 shows the precision for the top 5 sources in a plan. Again we can see that Greedy Select comes out the winner. The decrease in precision of plans generated for higher values of threshold can be explained from Figure 28. As can be seen, for larger values of threshold more number of leaf classes get pruned. A mediator query always maps to a particular leaf class. But for higher thresholds, the leaf classes are pruned and hence queries get mapped to higher level abstract classes. Therefore the statistics used to generate plans have lower accuracy and in turn generate plans with lower precision.

Altogether the experiments on these controlled datasets show that *StatMiner* uses the association mining based approach effectively to control the number of statistics required for data integration. An ideal threshold for a mediator relation would depend on the number and size of AV hierarchies. For our sample Books mediator, an ideal threshold for *StatMiner* would be around 0.75%, for both the hierarchies, where *StatMiner* effectively prunes a large number of small classes and yet the precision of plans generated is fairly high. We also bring forth the problems involved in trying to scale up the algorithm to larger hierarchies.

3.10. Discussion

In the previous sections of this chapter, we focussed on learning coverage and overlap statistics of selection and projection queries. The techniques described for selection queries can however be extended to join queries. In this section, we first define the concept of coverage and overlap statistics with respect to join queries, then we discuss how we can discover frequent query classes of join queries, and learn statistics for these classes.

For a join query $Q : -R_1, R_2, ..., R_n$ and a subgoal relation R_i in Q, we use $Q \cap R_i$ to denote the set of tuples from the subgoal R_i that appear in the final answer tuple set after joining with all other subgoals in Q. The *coverage* of a data source S with respect to Q and R_i, denoted by $P(S|Q \cap R_i)$, is the probability that a random tuple of $Q \cap R_i$ is present in source S. The *overlap* among a set $\widehat{S}$ of sources with respect to Q and R_i, denoted by $P(\widehat{S}|Q \cap R_i)$, is the probability that a random tuple of $Q \cap R_i$ is present in each source $S \in \widehat{S}$. The overlap statistics (or coverage when $\widehat{S}$ is a singleton) w.r.t. Q and R_i are computed using the following formula

$$P(\widehat{S}|Q \cap R_i) = \frac{N_{Q \cap R_i}(\widehat{S})}{N_{Q \cap R_i}}$$

Here $N_{Q \cap R_i}(\widehat{S})$ is the number of tuples in $Q \cap R_i$ that are in all sources of $\widehat{S}$, $N_{Q \cap R_i}$ is the total number of tuples in $Q \cap R_i$. we assume that the union of the contents of the available sources within the system covers 100% of all the subgoal relations. In other words, coverage and overlap are measured relative to the available sources.

We also define coverage and overlap with respect to a query class, C, of join queries and subgoal relation R_i rather than a single join query Q. The overlap of a source set $\widehat{S}$ (or coverage when $\widehat{S}$ is a singleton) w.r.t. a query class C and subgoal relation R_i can be computed using the following formula:

$$P(\widehat{S}|C \cap R_i) = \frac{\sum_{Q \in C} P(\widehat{S}|Q \cap R_i)P(Q)}{P(C)}$$

The coverage and overlap statistics w.r.t. a class C are eventually used to estimate the source coverage and overlap for all the queries that are mapped into C.

We classify all join queries with the same subgoal relations into a *join query class.* All the subgoal relations of the queries within a join query classes are considered as a single relation. For example, consider the following three queries Q_1, Q_2, and Q_3:

$$Q_1 : -R_1(A_1, A_2), R_2(A_2, A_3), A_1 = a$$

$$Q_2 : -R_1(A_1, A_2), R_2(A_2, A_3), A_3 = c$$

$$Q_3 : -R_1(A_1, A_2), R_2(A_2, A_3), A_1 = b, A_3 = d$$

Since they all have the same subgoal relations:$R_1(A_1, A_2)$ and $R_2(A_2, A_3)$, we can classify them into the same join query class. we use a single relation $R(A_1, A_2, A_3)$ to denote the join of $R_1(A_1, A_2)$ and $R_2(A_2, A_3)$, then Q_1, Q_2, and Q_3 can be rewritten as:

$$Q_1 : -R(A_1, A_2, A_3), A_1 = a$$

$$Q_2 : -R(A_1, A_2, A_3), A_3 = c$$

$$Q_3 : -R(A_1, A_2, A_3), A_1 = b, A_3 = d$$

As we can see these queries becomes selection queries over the new relation $R(A_1, A_2, A_3)$.

After we rewrite the queries within a join query class into selection queries, we can classify these queries based on their bound values and use similar techniques for selection queries to learn statistics for frequent query classes within the join query class[13].

We now discuss how to compute the coverage and overlap statistics of the discovered frequent classes within a join query class. Specifically, all the coverage and overlap statistics, $P(\widehat{S}|Q \cap R_i)$, of user queries will be computed and recorded in the query list the first time the queries are asked by users. After the frequent query classes are discovered, the statistics of each frequent class will be learned in the same way as for learning statistics for selection query classes.

3.11. Summary

In this chapter we motivated the need for automatically mining the coverage and overlap statistics of sources w.r.t. frequently accessed query classes for efficient query processing in a data integration scenario. We then presented a set of connected techniques that automatically generate attribute value hierarchies, efficiently discover frequent query classes

[13]If the join query class itself is not frequent enough, statistics for this class will not be learned. Instead, the statistics of each of the subgoal relations in the join queries within the class will be used to estimate their statistics.

and learn coverage and overlap statistics for only these frequent classes. We described the algorithmic details and implementation of our approach. We also presented an empirical evaluation of the effectiveness of our approach in the bibliography mediator *BibFinder*. Our experiments demonstrate that (i) We can systematically trade the statistics learning time and number of statistics remembered for accuracy by varying the frequent class thresholds. (ii) The learned statistics provide tangible improvements in the source ranking, and the improvement is proportional to the granularity of the learned statistics.

CHAPTER 4

JOINT OPTIMIZATION OF COVERAGE AND COST WITH MULTI-R

As we mentioned earlier, selecting high-quality plans in data integration requires the ability to consider the coverages offered by various sources, and form a query plan with the combination of sources that is estimated to be a high-quality plan given the cost-coverage tradeoffs of the user. In this chapter, we describe how *Multi-R* effectively uses the coverage and overlap statistics learned by *StatMiner*, and does joint optimization of coverage and cost of query plans in data integration.

The rest of the chapter is organized as follows. Section 4.1 uses a simple example to provide a brief survey of existing work on query optimization in data integration, as well as to motivate the need for our joint optimization approach. Section 4.2 discusses the syntax and semantics of the parallel query plans, the models for estimating the cost and coverage of parallel plans, and the specific methodology we use to combine cost and coverage into an aggregate utility metric. Section 4.3 describes two algorithms to generate parallel query plans and analyzes their complexity. Section 4.4 presents a comprehensive empirical evaluation which demonstrates that *Multi-R* can offer high utility plans (in terms of cost and coverage) for a fraction of the planning cost incurred by the existing approaches that

Sources	Relations	Coverage	Cost	Must bind attributes
S_{11}	**book**	70%	300	ISBN or title
S_{12}	**book**	50%	200	ISBN or title
S_{13}	**book**	60%	600	ISBN
S_{21}	**price-of**	75%	300	ISBN or retail-price
S_{22}	**price-of**	70%	260	ISBN or retail-price
S_{31}	**review-of**	70%	300	ISBN
S_{32}	**review-of**	50%	400	reviewer

Table 1. Statistics for the sources in the example system

use phased optimization with linear plans.

4.1. Motivation

Consider a simple mediator that integrates several sources that export information about books. Suppose there are three relations in the global schema of this system: **book**(*ISBN, title, author*), **price-of**(*ISBN, retail-price*), **review-of**(*ISBN, reviewer, review*). Suppose the system can access three sources: $\mathbf{S}_{11}$, $\mathbf{S}_{12}$, $\mathbf{S}_{13}$, each of which contain tuples for the **book** relation, two sources $\mathbf{S}_{21}$, $\mathbf{S}_{22}$ each of which contain tuples for the **price-of** relation, and two sources $\mathbf{S}_{31}$, $\mathbf{S}_{32}$ each of which contain tuples for the **review-of** relation. Individual sources differ in the amount of coverage they offer on the relation they export. Table 1 lists some representative statistics for these sources. We will assume that the coverage is measured in terms of the fraction of tuples of the relation in the global schema which are stored in the source relation, and cost is specified in terms of the average response time for a single source call. The last column lists the attributes that must be bound in each call to the source. To simplify matters, let us assume that the sources are "independent" in their coverage (in that the probability that a tuple is present in a given source is independent of the probability that the same tuple is present in another source). Consider the example query:

$$\begin{aligned}\mathbf{Q}(\textit{title,retail-price,review}) :- &\ \mathbf{book}(\textit{ISBN, title, author}),\\ &\ \mathbf{price\text{-}of}(\textit{ISBN, retail-price}),\\ &\ \mathbf{review\text{-}of}(\textit{ISBN, reviewer, review}),\\ &\ \textit{title}=\text{“Data Warehousing”, retail-price}<\$40.\end{aligned}$$

In the following, we briefly discuss the limitations of existing approaches in optimizing this query, and motivate our approach.

Bucket Algorithm [LRO96]: The bucket algorithm by Levy et al. [LRO96] will generate three buckets, each containing the sources relevant to one of the three subgoals in the query:

Bucket B(for **book**): $\mathbf{S}_{11}$**,** $\mathbf{S}_{12}$**,** $\mathbf{S}_{13}$

Bucket P(for **price-of**): $\mathbf{S}_{21}$**,** $\mathbf{S}_{22}$

Bucket R(for **review-of**): $\mathbf{S}_{31}$**,** $\mathbf{S}_{32}$

Once the buckets are generated, the algorithm will enumerate 12 possible plans ($= 3\times 2\times 2$) corresponding to the selection of one source from each bucket. For each combination, the correctness of the plan is checked (using containment checks), and executable orderings for each plan are computed. Note that the 6 plans that include the source S_{32} are not going to lead to any executable orderings since there is no way of binding the “reviewer” attribute as the input to the source query. Consequently, the set of plans output by the bucket algorithms are:

$$\mathrm{p}_1 = (S_{11}^{fbf} \bowtie S_{21}^{bf}) \bowtie S_{31}^{bff},$$

$$\mathrm{p}_2 = (S_{11}^{fbf} \bowtie S_{22}^{bf}) \bowtie S_{31}^{bff},$$

$$\mathrm{p}_3 = (S_{12}^{fbf} \bowtie S_{21}^{bf}) \bowtie S_{31}^{bff},$$

$$\mathrm{p}_4 = (S_{12}^{fbf} \bowtie S_{22}^{bf}) \bowtie S_{31}^{bff},$$

$$\mathrm{p}_5 = (S_{21}^{fb} \bowtie S_{13}^{bff}) \bowtie S_{31}^{bff},$$

$$\mathrm{p}_6 = (S_{22}^{fb} \bowtie S_{13}^{bff}) \bowtie S_{31}^{bff}$$

where, the superscripts "f" and "b" are used to specify which attributes are bound in each source call. We call these plans "linear plans" in the sense that they contain at most one source for each of the relations mentioned in the query. Once the feasible logical plans are enumerated, the approach in [LRO96] consists of finding "feasible" execution orders for each of the logical plans, and executing *all* the plans. While this approach is guaranteed to give maximal coverage, it is often prohibitively expensive in terms of both planning and execution cost. In particular, for a query with n subgoals, and a scenario where there are at most m sources in the bucket of any subgoal, the worst case complexity of this approach (in terms of planning time) is $O(m^n n^2)$, as there can be m^n distinct linear plans, and the cost of finding a feasible order for them using the approach in [LRO96] is $O(n^2)$.

Executing top N Plans: More recent work [FKL97; NLF99; DH02] tried to make-up for the prohibitive execution cost of the enumeration strategy used in [LRO96] by first ranking the enumerated plans in the order of their coverage (or more broadly "quality"), and then executing the top N plans, for some arbitrarily chosen N. The idea is to identify the specific plans that are likely to have high coverage and execute those first.

In our example, these procedures might rank $\mathrm{p}_1 = (S_{11}^{fbf} \bowtie S_{21}^{bf}) \bowtie S_{31}^{bff}$ as the best plan (since all of the sources have the highest coverage among sources in their buckets), and then rank $\mathrm{p}_6 = (S_{22}^{fb} \bowtie S_{13}^{bff}) \bowtie S_{31}^{bff}$, as the second best plan as it contains the sources with highest coverages after executing the best plan p_1.

The problem with this type of approach is that *the plans that are ranked highest in terms of coverage may not necessarily provide the best tradeoffs in terms of execution cost.* In our example, suppose source S_{22} stores 1000 tuples with attribute value retail-price less than \$40, then in plan p_6 we have to query S_{13}, the costliest among the accessible sources, a thousand times because of its binding pattern restriction. The total cost of this plan will thus be more than 6×10^5. In contrast, a lower ranked plan such as p$_4$ ($= (S_{12}^{fbf} \bowtie S_{22}^{bf}) \bowtie S_{31}^{bff}$) may cost significantly less, while offering coverage that is competitive with that offered by p_6. For example, assuming that source S$_{12}$ maintains 10 independent ISBN values for title="Data Warehousing", the cost of p_4 may be less than 5800. In such a scenario, most users may prefer executing the plan p$_4$ first instead of p$_6$ to avoid incurring the high cost of executing plan p$_6$.

The lesson from the example above is that if we want to get a plan that can produce higher quality results with limited cost, it is critical to consider execution costs *while* doing source selection (rather than *after the fact*). In order to take the cost information into account, we have to consider the source-call ordering during planning, since different source-call orders will result in different execution costs for the same logical plan. In other words, we have to jointly optimize source-call ordering and source selection to get good query plans.

The Need for Parallel Plans: Once we recognize that the cost and coverage need to be taken into account together, we argue that it is better to organize the query planning in terms of "which sources should be called for supporting each subgoal" rather than in terms of "which linear plans have the highest cost/quality tradeoffs." To this end, we introduce the notion of a "parallel" plan, which is essentially a sequence of source sets. Each source set corresponds to a subgoal of the query, such that the sources in that set export that subgoal (relation). The sources in individual source sets can be accessed in parallel (see Figure 32).

In our continuing example, looking at the six plans generated by the bucket algorithm we can see that the first four plans p$_1$, p$_2$, p$_3$ and p$_4$ have the same subgoal order (the order of the subgoals of the sources in the plan): *book* $\rightarrow$*price-of* $\rightarrow$ *review-of*; while the other two plans p_5, p_6 have the subgoal order:*price-of* $\rightarrow$*book* $\rightarrow$ *review-of.* So we can use the following two parallel plans to give all of the tuples that the six plans give in this example:

$$p_1'=((S_{11}^{fbf} \cup S_{12}^{fbf}) \bowtie (S_{21}^{bf} \cup S_{22}^{bf}))\bowtie S_{31}^{bff},$$

$$p_2'=((S_{21}^{fb} \cup S_{22}^{fb}) \bowtie S_{13}^{bff})\bowtie S_{31}^{bff}$$

These plans access all the sources related to a given subgoal of the query in parallel (see Section 4.2 for a more detailed description of their semantics). An important advantage of these plans over linear plans is that they avoid the significant redundant computation inherent in executing all feasible linear plans separately. In our example, plan p_1 and p_2 will both execute source queries S_{11}^{fbf} and S_{31}^{bff} with the same binding patterns. In contrast, $p_1{}'$ avoids this redundant access.[1]

The Need for *searching* in the space of parallel plans: One remaining question is whether we should search in the space of parallel plans directly, or search in the space of linear plans and post-process the linear plans into a set of equivalent parallel plans. An example of the post-processing approach may be one which generates top N plans using methods similar to those outlined in [DH02] and then parallelizes them. However such approaches in general are not guaranteed to give cost-coverage tradeoffs that are attainable by searching in the space of parallel plans because: (i) the cost of generating a single high-quality parallel plan can be significantly lower than the cost of enumerating, rank-ordering the top N linear plans and post-processing them and (ii) since the post-processing approaches separate the cost and coverage considerations, the utility of the resulting plans can be arbitrarily far from the optimum.

[1]Notice that here we are assuming that the linear plans are all executed independently of one another. A related issue is the optimal way to execute a union of linear plans–they can be executed in sequence, with cached intermediate results (which will avoid the redundant computation, but increases the total execution time), or executed in parallel (which reduces the execution time but incurs the redundant accesses). These two options are really special cases of the more general option of post-processing the set of linear plans into a minimal set of parallel plans and executing them (see below).

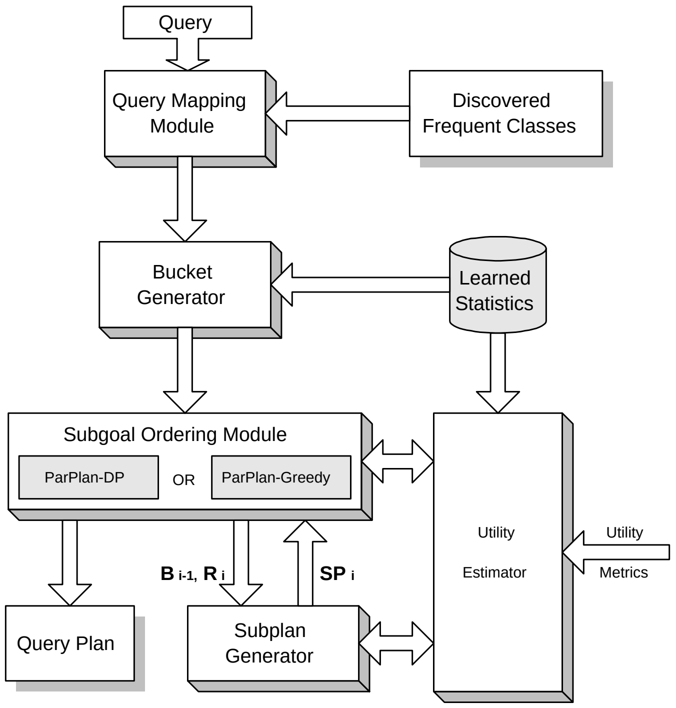


Figure 31. *Multi-R* architecture

Moreover, we will see that the main possible objection to searching in the space of parallel plans–that the space of parallel plans may be much larger than the space of linear plans–turns out to be a red herring. Specifically, our approach involves searching in the space of subgoal orders, and for each subgoal order efficiently generating a high-quality parallel plan. This approach winds up adding very little additional planning overhead over that of searching in the space of linear plans, and even this overhead is more than made up for by the fact that we avoid the inefficiencies of phased optimization.

The joint optimization approach described and motivated in the foregoing has been implemented in *Multi-R*. Figure 31 shows the architecture of *Multi-R*. For a user query, *Multi-R* first maps it to a set of frequent query classes and estimates the coverage and overlap statistics for the query using the techniques described in Section 3.7. Using these statistics the bucket generator generates a bucket, for each subgoal of the query, which contains all the relevant sources exporting the subgoal relation. *Multi-R* searches in the space of "parallel" query plans. The partial (parallel) plans are evaluated in terms of a general "utility" metric, that takes both cost and coverage of the plan into account. The search for query plans is done in terms of two interleaved processes: the first is a search conducted in the space of subgoal orders, and the second is a procedure that provides a high-quality subplan for a given subgoal, in the context of the current partial plan. The first process can be done either with a dynamic programming style search or a greedy search. For the second one, we use a greedy algorithm (that is nevertheless optimal when certain restrictions are met on subgoal independence). In the following two sections, we discuss the cost models we use for evaluating parallel query plans, and the search algorithms used to generate query plans with highest utility.

4.2. Parallel Query Plans

In this section, we first formally define the concept of a query and a parallel query plan. we then discuss how to compute the cost and coverage of a parallel plan. At last we discuss how to combine the cost and coverage into a single utility metric according to user preferences.

4.2.1. Queries and Parallel Query Plans.

Let's assume $R_1(\overline{X}_1)$, $R_2(\overline{X}_2)$, ..., $R_n(\overline{X}_n)$ are mediated schema relations, a query in our data integration system has the form: $Q(\overline{X})$:- $R_1(\overline{X}_1)$, $R_2(\overline{X}_2)$, ..., $R_n(\overline{X}_n)$. The atom $Q(\overline{X})$ is called the *head* of the datalog rule, and the atoms $R_1(\overline{X}_1)$, $R_2(\overline{X}_2)$, ..., $R_n(\overline{X}_n)$ are the *subgoals* in the *body* of the rule. The tuples $\overline{X}$, $\overline{X}_1$, $\overline{X}_2$, ..., $\overline{X}_n$ contain either variables or constants, and we need $\overline{X} \subseteq \overline{X}_1 \cup \overline{X}_2 \cup ... \cup \overline{X}_n$ for the query to be safe.

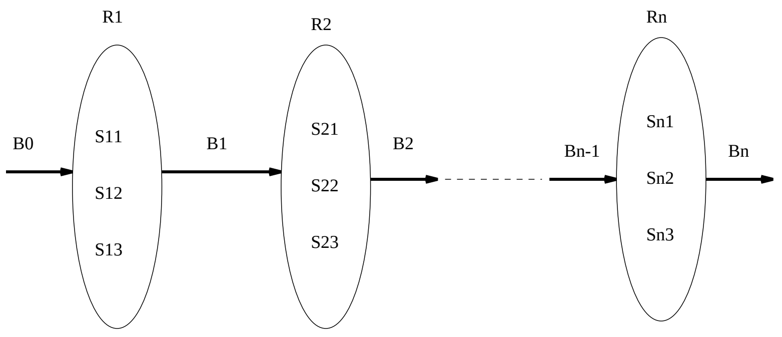


Figure 32. A parallel query plan

A parallel query plan p has the form

$$p = (\ (...(sp_1 \bowtie sp_2) \bowtie ...) \bowtie sp_{n-1}) \bowtie sp_n,$$

$$\text{where } sp_i = (S_{i1} \cup S_{i2} \cup ... \cup S_{im_i})$$

Here sp$_i$ is a subplan and S$_{ij}$ is a source relation corresponding to the i^{th} subgoal of the subgoal order used in the query plan. The semantics of subplan sp$_i$ are that it queries its sources in parallel and unions the results returned from the sources. The semantics of plan p are that it joins the results of the successive subplans to answer the query.

To clarify this process more, we need the concept of *binding relations*,[2] which are intermediate relations that keep track of the partial results of executing the first k subplans of the query plan. Given a query plan of n subgoals in the order of R_1, R_2, ..., R_n, we define a corresponding sequence of $n+1$ *binding relations* B_0, B_1, B_2, ..., B_n (see Figure 32). B_0 has the set of variables bound in the query as its schema, and has as its instance a single tuple, denoting the bindings specified in the query. The schema of B_1 is the union of the schema of B_0 and the schema of the mediated relation of R_1. Its instance is the join of B_0 and the union of the source relations in the subplan of R_1. Similarly we define B_2 in terms

[2]The idea of binding relations is first introduced in [YLUG99] for linear query plans where each subgoal of the query has only one source relation. We use a generalization of this idea to parallel plans.

of B_1 and the mediated relation of R_2, and so on. The answer to the (conjunctive) query is defined by a projection operation on B_n.

4.2.2. Cost and Coverage Models. The main aim of this section is to describe the models we use to estimate the execution cost and coverage of (parallel) query plans, and how we combine the cost and coverage components into an aggregate utility metric for the plan.

Source Statistics: For a source S defined over the attributes $A = \{A_1, A_2, ..., A_m\}$ and the mediated schema defined in the data integration system as $R_1, R_2, ..., R_n$, we currently assume the following statistical data:

1. For each attribute A_i, its length, and for each attribute A_i in source relation S, the number of distinct values of A_i;
2. For each source: the number of tuples, the feasible binding patterns, the local delay time to process a query, the bandwidth between the source and the integration system and the initial set up latency;
3. For each mediated relation R_j, its coverage in the source S, denoted by $P(S|R_j)$, for example, $P(S|author) = 0.8$ denotes that source S stores 80% of the tuples of the mediated relation $author(name, title)$ of all the sources in the data integration system. Following [NLF99, FKL97], we also make the simplifying assumption that the sources are "independent" in that the probability that a tuple is present in source S_1 is independent of the probability that the same tuple is present in S_2.

These assumptions are in line with the types of statistics used by previous work [LRO96, NLF99]. Techniques for learning response time statistics through probing are discussed in [GRZ$^+$00], while those for learning coverage statistics are discussed in the Chapter 3.

Estimating the Cost of a parallel plan: In this dissertation, we will estimate the cost of a parallel plan purely in terms of its execution time. We will also assume that the execution time is dominated by the tuple transfer costs, and thus ignore the local processing costs at the mediator (although this assumption can be relaxed without affecting the advantages of our approach). Thus the execution costs are computed in terms of the response times offered by the various sources that make up the (parallel) plan. The response time of a source is proportional to the number of times that source is called, and the expected number of tuples transferred over each call. Since the sources have binding pattern limitations, and the set of feasible source calls depend on the set of call variables that can be bound, both the number of calls and the number of tuples transferred depend on the value of the *binding relation* preceding the source call.

Specifically, suppose we have a plan p with the subplans $\{sp_1, sp_2, ..., sp_n\}$. The cost of p is given by:

$$cost(p) \doteq \sum_i responseTime(sp_i)$$

The response time of each subplan $sp_i (= \{S_{i_1}, S_{i_2}, ..., S_{i_m}\} \in p)$ is computed in terms of the response times of the individual sources that make up the subplan. Since the sources are processed in parallel, the cumulative response time is computed as the sum of the maximum response time of all the sources in the subplan and a fraction of the total response time of the sources in the subplan:

$$responseTime(sp_i) = max_{j \in [1,m]}\{responseTime(S_{i_j}, B_{i-1})\} + \beta \times \textstyle\sum_{j \neq max_{RT}} responseTime(S_{i_j}, B_{i-1})$$

where β is a weight factor between 0 and 1, which depends on the level of parallelism assumed to be supported by the system and the network. $\beta = 0$ means that the system allows full parallel execution of all the sources queries, while $\beta = 1$ means that all the source queries have to be executed strictly sequentially. Notice that the response time of each source is being computed in the context of the binding relation preceding that source call. For a source S under the binding relation B, we have

$$\begin{aligned} responseTime(S, B) = {} & msgDelay(S) * msgs(S, B) + \\ & bytes(S, B)/localDelay(S) + \\ & bytes(S, B)/bandWidth(S) \end{aligned}$$

where $msgs(S, B)$ is the number of separate calls made to the source S under the binding relation B, and $bytes(S, B)$ denotes the total bytes sent back by the source S in response to these calls.

Estimating the Coverage of a parallel plan: For a plan $p = \{sp_1, sp_2, ..., sp_n\}$, the coverage of p will depend on the coverages of the subplans sp_i in p and the join selectivity factors of the subgoals and sources associated with these subplans. Let R_{sp_i} be the corresponding subgoal of the subplan sp_i=$\{S_{i_1}, S_{i_2}, ..., S_{i_m}\}$. We use $SF_J(B_{i-1}, sp_i)$ to denote the join selectivity factor between the sources within the i^{th} subplan and the binding relation resulting from joining the first $i-1$ subplans, and $SF_J(\widehat{B}_{i-1}, R_{sp_i})$ to denote the join selectivity factor between the i^{th} subgoal relation and the binding relation resulting from joining the first $i-1$ *subgoal relations.* Coverage of the plan p can be computed as:

$$coverage(p) = \frac{\prod_{i=1}^{n} [card(sp_i) \times SF_J(B_{i-1}, sp_i)]}{\prod_{i=1}^{n} [card(R_{sp_i}) \times SF_J(\widehat{B}_{i-1}, R_{sp_i})]}$$

If we assume that the subplans cover their respective relations uniformly (which is likely to be the case as the sizes of the subplans and their coverages increase), then we have

$$SF_J(B_{i-1}, sp_i) = SF_J(\widehat{B}_{i-1}, R_{sp_i}).$$

This, together with the fact that $\frac{card(sp_i)}{card(R_{sp_i})}$ is just the definition of $P(sp_i|R_{sp_i})$, simplifies the expression for coverage of p to

$$coverage(p) = \prod_{i=1}^{n} [P(sp_i | R_{sp_i})]$$

The coverage of a subplan itself can be written in terms of the coverages provided by the individual sources exporting that relation:

$$\begin{aligned} P(sp_i|R_{sp_i}) &= P(\cup_{S_{i_j} \in sp_i} S_{i_j} | R_{sp_i}) \\ &= P(S_{i_1}|R_{sp_i}) + P(S_{i_2} \wedge \neg S_{i_1}|R_{sp_i}) + ...+ \\ & P(S_{i_m} \wedge \neg S_{i_1} \wedge ... \wedge \neg S_{i_{m-1}}|R_{sp_i}) \end{aligned}$$

As mentioned earlier, we assume that the contents of the sources are independent of each other. That is, the presence of a tuple in one source does not change the probability that the tuples also belongs to another source. Thus, the conjunctive probabilities can all be computed in terms of products. E.g.

$$P(S_{i_2} \wedge \neg S_{i_1}|R_{sp_i}) = P(S_{i_2}|R_{sp_i}) * (1 - P(S_{i_1}|R_{sp_i}))$$

4.2.3. Combining Cost and Coverage. The main difficulty in combining the cost and the coverage of a plan into a utility measure is that, as the length of a plan (in terms of the number of subgoals covered) increases, the cost of the plan increases additively, while the coverage of the plan decreases multiplicatively. In order to make these parameters combine well, we take the sum of the logarithm of the coverage component and the negative of the cost component:[3]

$$utility(p) = w * log(coverage(p)) - (1 - w) * cost(p)$$

[3]We adapt this idea from [C01] for combining the cost and quality of Multimedia database query plans, where the cost also increases additively and the quality (such as precision and recall) decreases multiplicatively when the number of predicates increases.

The logarithm ensures that the coverage contribution of a set of subgoals to the utility factor will be additive. The user can vary w from 0 to 1 to change the relative weight given to cost and coverage.[4]

4.3. Generating Query Plans

The algorithms presented in this section aim to find a high-quality parallel plan–i.e., the parallel plan with high utility. Our basic plan of attack involves considering different feasible subgoal orderings of the query, and for each ordering, generating a parallel plan that has the highest utility. To this end, we first consider the issue of generating a high-quality plan for a given subgoal ordering.

Given the semantics of parallel plans (see Figure 32), this involves finding a high-quality "subplan" for a subgoal relation under a given binding relation. We provide an algorithm for doing this in Section 4.3.1. We then tackle the question of searching the space of subgoal orders. For this, we develop a dynamic programming algorithm (Section 4.3.2) as well as a greedy algorithm (Section 4.3.3).

4.3.1. Subplan Generation. The algorithm *CreateSubplan* shown in Algorithm 1 computes a high-quality subplan for a subgoal R, given the statistics about the m sources $S_1, S_2, ..., S_m$ that export R, and the binding relation at the end of the current (partial) plan, *CreateSubplan* first computes the utility of all the sources in the bucket, and then sorts the sources according to their utility value. Next the algorithm adds the sources from the sorted bucket to the subplan one by one, until the utility of the current subplan becomes

[4]In the actual implementation we scale the coverage appropriately to handle the discontinuity at 0, and use normalization to make the contribution from the coverage component to be in the same range as that from the cost component.

less than the utility of the previous subplan. We use the models discussed in Section 4.2.2 to calculate the utility (cost and coverage) of the subplans.

Algorithm 1 CreateSubplan

```
 1: input: $B$: the binding relation; $R$ : the subgoal in the query
 2: output: $sp$ : the best plan
 3: begin
 4: $sp \leftarrow \{\}$
 5: $Bucket \leftarrow$ the Bucket for the subgoal $R$;
 6: for each source $s \in Bucket$ do
 7:    if ($s$ is feasible under $B$) then
 8:       $utility(s) = w * log(coverage(s)) -$
                  $(1 - w) * responseTime(s, B)$;
 9:    else
10:       $remove$ $s$ from $Bucket$
11:    end if
12: end for
13: sort the sources in $Bucket$ in decreasing order of their utility(s);
14: $s \leftarrow$ the first source in the sorted $Bucket$;
15: while ( $s$ != null) and ($utility(sp + \{s\}) > utility(sp)$ ) do
16:    $sp \leftarrow sp + \{s\}$
17:    $s \leftarrow$ the next source in the $Bucket$;
18: end while
19: return sp;
20: end
```

Although the algorithm has a greedy flavor, the subplans generated by this algorithm can be shown to be optimal if the sources are conditionally independent [FKL97] (i.e., the presence of an object in one source does not change the probability that the object belongs to another source). Under this assumption, the ranking of the sources according to their coverage and cost will not change after we execute some selected sources.

The running time of the algorithm is dominated by line 15, which is executed m times, taking $O(m)$ time in each loop for computing the utility of the subplan (under the source independence assumption). Thus the algorithm has $O(m^2)$ complexity.

4.3.2. A Dynamic Programming Approach for Parallel Plan Generation.

In the following we introduce a dynamic programming-style algorithm called *ParPlan-DP* which extends the traditional System-R style optimization algorithm to find a high-quality parallel plan for the query. The basic idea is to generate the various permutations of the subgoals, compute a high-quality parallel plan (in terms of utility) for each permutation, and select the best among these. While our algorithm is related to a traditional system-R style algorithm, as well as its recent extension to handle binding pattern restrictions (but without multiple overlapping sources), given in [FLMS99], there are some important differences:

1. *ParPlan-DP* does source selection and subgoal ordering together according to our utility model for parallel plans; while the traditional System-R and [FLMS99] just need to pick a single best subgoal order according to the cost model.

2. *ParPlan-DP* has to estimate attribute sizes of the binding relations for partial parallel plans, where there are multiple sources for a single subgoal. So we have to take the overlap of sources in the subplan into account to estimate the sizes of each of the attributes in the binding relation.

3. *ParPlan-DP* needs to remember all the best partial plans for every subset of one or more of the n subgoals. For each subset, it stores the following information: (i) the best plan for these subgoals; (ii) the binding relation of the best plan; and (iii) the cost, coverage and utility of the best plan. In contrast, a traditional system-R style optimizer need only track the best plan, and its cost [SACL79].

The subgoal permutations are produced by the dynamic construction of a tree of alternative plans. First, high-quality plans for single subgoals are computed, followed by the plans for pairs and larger subsets of subgoals, until the plan for n subgoals is computed. When we have the plan for any i subgoals, we can find the plan for $i+1$ subgoals by using the results of first i subgoals and finding the best subplan for the i+1^{th} subgoal under the

Algorithm 2 *ParPlan-DP*

```
1: Input: $BUCKETS$ : Buckets for the n subgoals;
2: output: $p$ : the best plan
3: begin
4: $S \leftarrow \{\}$; {a queue to store plans;}
5: $p_0.plan \leftarrow \{\}$; {$p_0$: the initial node}
6: $p_0.B \leftarrow B_0$; {the binding relation of $p_0$: $B_0$}
7: $p_0.R \leftarrow \{\}$; {the selected subgoals of $p_0$: empty}
8: $p_0.utility \leftarrow -\infty$; {the utility of $p_0$: negative infinity}
9: $S \leftarrow S + \{p_0\}$;
10: $p \leftarrow$ pop the first plan from $S$;
11: while ($p \neq null$) and (# of subgoals $p.R < n$) do
12:   for each feasible subgoal $R_i (\in BUCKETS$ and $\notin p.R)$ do
13:     make a new plan $p'$ ;
14:     $sp \leftarrow CreateSubplan(p.B, R_i)$;
15:     $p'.plan \leftarrow p.plan + sp$;
16:     $m \leftarrow$ # of sources in sp;
17:     $p'.B \leftarrow p.B \bowtie (\bigcup_{i=1}^{m} S_i)$; $\{S_i \in sp\}$
18:     $p'.R \leftarrow p.R + \{R_i\}$;
19:     $p'.utility \leftarrow utility(p')$;
20:     if ( $\exists p_1 \in S$ ) and ($p_1.R$ commutatively equals $p'.R$) and ($p'.utility > p_1.utility$) then
21:       remove $p_1$ from $S$ and push $p'$ into $S$
22:     else if ($p'.utility \leq p_1.utility$) then
23:       if ($w = 1$) and ($p'.coverage = p_1.coverage$) and ($p'.cost \leq p_1.cost$) then
24:         remove $p_1$ from $S$ and push $p'$ into $S$
25:       else
26:         ignore $p'$
27:       end if
28:     else
29:       push $p'$ into $S$;
30:     end if
31:   end for
32:   $p \leftarrow$ pop the first plan from $S$;
33: end while
34: return $p$;
35: end
```

binding relation given by the subplans of the first i subgoals. In practice, the algorithm does not need to generate all possible permutations. Permutations involving subgoals without any feasible source queries are eliminated.

Complexity: The worst case complexity of query planning with *ParPlan-DP* is is $O(2^n m^2)$, where n is the number of subgoals in the query and m is the number of sources exporting each subgoal. The 2^n factor comes from the complexity of traditional dynamic programming, and the m^2 factor comes from the complexity of *CreateSubplan*.

We also note that our way of searching in the space of parallel plans does not increase the complexity of our query planning algorithm significantly. In fact, our $O(2^n m^2)$ complexity compares very favorably to the complexity of the linear plan enumeration approach described in [LRO96], which will be $O(m^n n^2)$, where m^n is the number of linear plans that can be enumerated, and n^2 is the complexity of the greedy algorithm they use to find the feasible execution order for each linear plan. This is despite the fact that the approach in [LRO96] is only computing *feasible* rather than optimal execution orders (the complexity would be $O(m^n 2^n)$ if they were computing optimal orders).

4.3.3. A Greedy Approach. We noted that *ParPlan-DP* already has better complexity than the linear plan enumeration approaches. Nevertheless, it is exponential in the number of query subgoals. In order to get a more efficient algorithm, we need to trade the optimality guarantees for performance. We introduce a greedy algorithm *ParPlan-Greedy* (see Algorithm 3) which gets a plan quickly at the expense of optimality.

This algorithm gets a feasible execution plan by greedily choosing the subgoal whose subplan can increase the utility of the current plan maximally. The subplan for that chosen subgoal is added to the plan, and the procedure is repeated until every subgoal has been covered.

The worst-case running time of *ParPlan-Greedy* is $O(n^2 m^2)$, where n is the number of subgoals in the query, and m is the number of sources per subgoal.

Theoretically, *ParPlan-Greedy* may produce plans that are arbitrarily far from the true optimum, but we shall see in Section 4.4 that its performance may be quite fair in practice.

Algorithm 3 ParPlan-Greedy

```
1: Input: $BUCKETS$ : Buckets with n subgoals;
2: output: $p$ : the best plan
3: begin
4: $B \leftarrow B_0$;
5: $UCS \leftarrow$ all n subgoals in the query;
6: $p \leftarrow \{\}$;
7: while ($UCS \neq \{\}$) do
8:    for each feasible subgoal $R_i (\in UCS)$ do
9:       $sp_i \leftarrow CreateSubplan(B, R_i)$;
10:   end for
11:   $sp_{max} \leftarrow$subplan which will maximize $utility(p + sp_i)$\{If $w = 1$, among the subplans with the highest
      coverage, we choose the subplan with cheapest cost.\};
12:   $p \leftarrow p + sp_{max}$;
13:   $UCS \leftarrow UCS - \{R_{max}\}$;
14:   $m \leftarrow$ # of sources in $sp_{max}$;
15:   $B \leftarrow B \bowtie (\bigcup_{i=1}^{m} S_i)$; $\{S_i \in sp_{max}\}$
16: end while
17: return $p$;
18: end
```

4.4. Empirical Evaluation

We have implemented the query planning algorithms described in this dissertation. In this section, we describe the results of a set of simulation studies that we conducted with these algorithms. The goals of the study are: (i) to compare the planning time and estimated quality of the solutions provided by our algorithms with the approaches that enumerate and execute all linear plans, (ii) to demonstrate that our algorithms are capable of handling a spectrum of desired cost-quality tradeoffs, and (iii) to compare the performance of *ParPlan-DP* and *ParPlan-Greedy*. Towards the first goal, we implemented the approach described in [LRO96]. This approach enumerates all linear plans, finds feasible execution orders for all of them, and executes them.

The experiments were done with a set of simulated sources. The sources are specified solely in terms of the source statistics. We used 206 artificial data sources and 10 mediated relations covering all these sources. The statistics for these sources were generated randomly, 60% sources have coverage of $20\% - 40\%$ of their corresponding mediated relations, 20%

sources have coverage of 40%−80%, 10% sources have coverage below 20%, and 10% sources have coverage above 80%. 90% of the sources have binding pattern limitations. We also set the response time statistics of these sources to represent both slow and fast sources: 20% of sources have a high response time, 20% of them have low response time, and 60% of them have a medium response time. The source statistics are used to estimate the costs and coverages of the plans, as described in Section 4.2. The queries used in our experiments are hybrid queries with both chain query and star query features [PL00], and their subgoals have 2-3 attributes. For example,

$$Q(A_1, A_4, A_7) : -R_1(A_1, A_2, A_3), R_2(A_2, A_3, A_4), R_3(A_3, A_5, A_6), R_4(A_6, A_7), A_4 = x0.$$

The comparison between our approach and that in [LRO96] will be influenced by the parameter β. When β is close to 1, the amount of parallelism supported is low. This is particularly hard on the approach in [LRO96] as all the linear plans have to be essentially sequentially executed. Because of this, in all our simulations (except those reported in Figure 36), we use $\beta = 0$ as the parameter to compute the response times as this provides the maximum benefit to the algorithms in [LRO96], and thus establishes a lower bound on the improvements offered by our approach in comparison. All our simulations were run under JDK 1.2.2 on a SUN ULTRA 5 with 256Mb of RAM.

Planning time comparison: Figure 33 and Figure 34 compare the planning time for our algorithms with the approach in [LRO96]. In Figure 33, we keep the number of sources per subgoal constant at 8, and vary the number of subgoals per query from 1 to 10. In Figure 34, we keep the number of subgoals constant at 3, and vary the number of sources per subgoal from 5 to 50. The planning time for [LRO96] consists of the time taken to produce all the linear plans and find a feasible execution order for each plan using the greedy approach in [LRO96], while the time for our algorithms consists of the time taken to construct and return their best parallel plan. We see right away that both our algorithms incur significantly lower plan generation costs than the decoupled approach used in [LRO96]. We also note that *ParPlan-Greedy* scales much better than *ParPlan-DP* as expected.

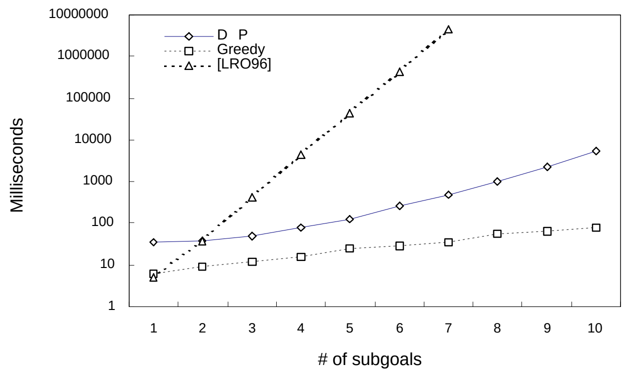


Figure 33. Variation of planning time with the query size (when the the number of relevant sources per subgoal is held constant at 8). X axis plots the query size while Y axis plots the planning time.

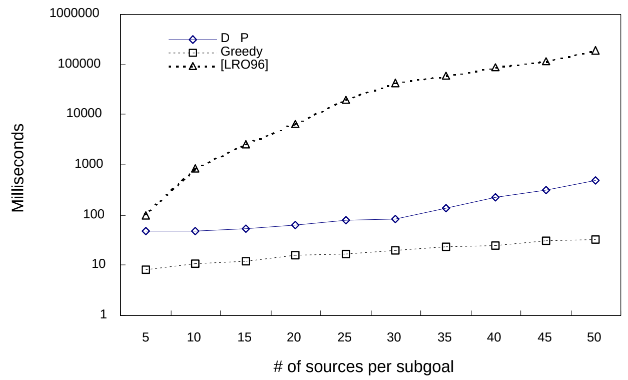


Figure 34. Variation of planning time with number of relevant sources per subgoal (for queries of size 3). X axis plots the query size while Y axis plots the planning time.

Quality comparison: In Figure 35, we plot the estimates of the cost and coverage of plans generated by *ParPlan-DP* as a percentage of the corresponding values of the plans given by the algorithm in [LRO96]. The cost and coverage values for the plans generated by each of the approaches are estimated from the source statistics, using the methods described in Section 4.2. We use queries with 4 subgoals and each subgoal with 8 sources. Notice that the algorithms in [LR096] do not take account of the relative weighting between cost and coverage. So the cost and coverage of the plans produced by this approach remains the same for all values of w. We note that the best plan returned by our algorithm has a pretty high estimated coverage (over 80% of the coverage for w over 0.4) while incurring cost that is below 2% of that incurred by [LRO96]. Note also that even though our algorithm seems to offer only 20% of the coverage offered by [LRO96] at w=0.1, this makes sense given that at w=0.1, the user is giving 9 times more weight to cost than coverage (and the approach of [LRO96] is basically ignoring this user preference and attempting full coverage).[5] In Figure 36, we compare the execution cost of plans given by our approach with that given by [LRO96] with different values of β in the response time estimation. As we can see the bigger the β, the better our plan execution cost relative performance. This is because, for any β larger than 0, the cost model will take into account the cost of redundant queries, which will further worsen the execution cost of the [LRO96].

[5]It is interesting to note that the execution cost for our approach turns out to be better than that of [LRO96] even when $w = 1$, when both approaches are forced to access all sources to maximize coverage. Since we kept $\beta = 0$, one might think that the execution costs should be same for this particular case. The reason our approach winds up having better execution cost even in this scenario is that it finds the best execution order of the parallel plan. In contrast, the [LRO96] approach just finds a feasible execution order for its plans. Because there are so many linear plans, the probability that one of them will wind up getting a feasible plan with high execution cost is quite high.

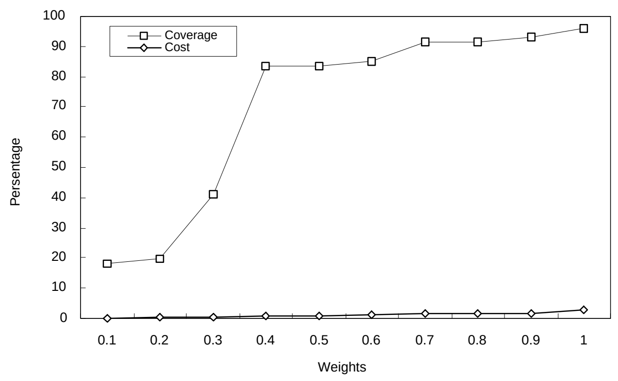


Figure 35. Comparing the quality of the plans generated by *ParPlan-DP* algorithm with those generated by [LRO96] (for queries of 4 subgoals), while the weight in the utility measure is varied. X axis shows the weight value in the utility measure and Y axis plots the cost and coverage of our algorithm expressed as a percentage of the cost and coverage provided by [LRO96].

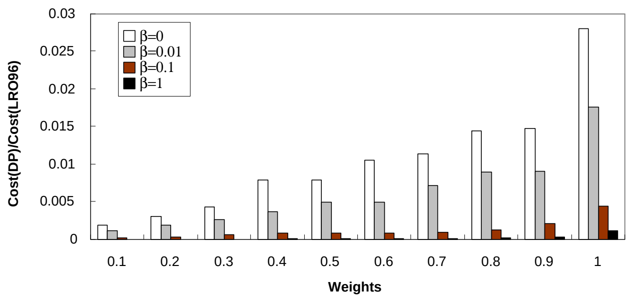


Figure 36. Ratio of the execution cost of the plans given by *ParPlan-DP* to that given by [LRO96], for a spectrum of weights in the utility metric and parallelism facter β in the response time estimate. X axis varies the coverage-cost tradeoff weights used in the utility metric, and Y axis shows the ratio of execution costs for different β.

Comparing the greedy and exhaustive approaches: In order to compare *ParPlan-*

DP and *ParPlan-Greedy* in terms of the plan quality, we experimented with queries with 4 subgoals and each subgoal with 8 sources, while the utility function is varied from being biased towards cost to being biased towards coverage. Figure 37 shows the utility of the best plan produced by *ParPlan-Greedy* as a percentage of the utility of the plan produced by *ParPlan-DP*. We observe, as expected, that the utility of plans given by *ParPlan-DP* is better than that of the *ParPlan-Greedy* with small initial weight (corresponding to a bias towards cost), with the ratio tending to 1 for larger weights (corresponding to a bias towards coverage). It is also interesting to note that at least in these experiments, the greedy algorithm is always producing plans that are within 70% of the utility of those produced by *ParPlan-DP*.

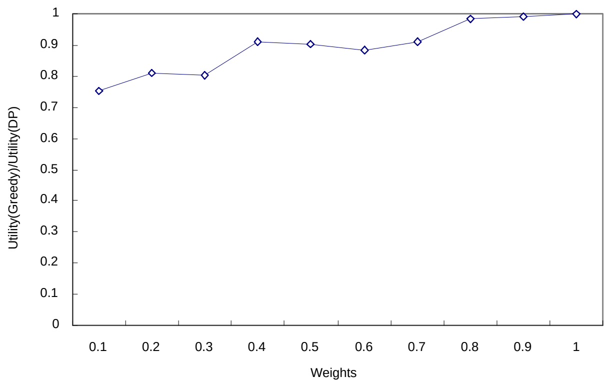


Figure 37. Ratio of the utility of the plans given by *ParPlan-Greedy* to that given by *ParPlan-DP* for a spectrum of weights in the utility metric. X axis varies the weight used in the utility metric, and Y axis shows the ratio of utilities.

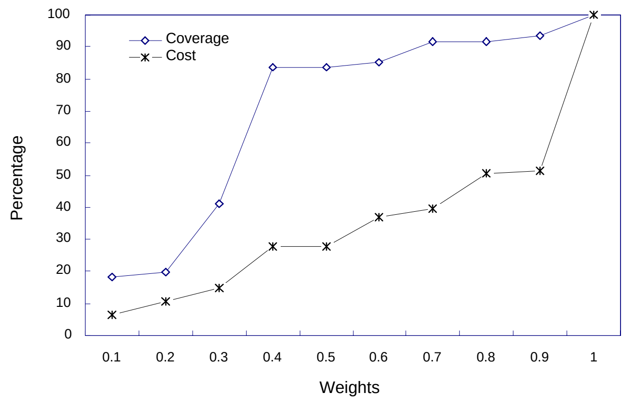


Figure 38. Comparing the Coverage and cost of the plans found by *ParPlan-DP* by using different weights in Utility function, on queries of 4 subgoals. X axis varies the weights in the utility function, while the Y axis shows the cost and coverage as a percentage of the cost and coverage offered by our ParPlan-DP with weight=1.

Ability to Handle a spectrum of cost-coverage tradeoffs: Our final set of experiments was designed to showcase the ability of our algorithms to handle a variety of utility functions and generate plans optimized for cost, coverage or a combination of both. Figure 38 shows how the coverage and the cost of plans given by our *ParPlan-DP* changes when the weight of the coverage in the utility function is increased. We observe that as expected both the coverage and the cost increase when we try to get higher coverage. We can also see that for the particular query at hand, there is a large area in which the cost increases slowly while the coverage increases more rapidly. An intriguing possibility offered by plots like this is that if they are done for representative queries in the domain, the results can be used to suggest the best initial weightings–those that are likely to give high coverage plans with relatively low cost–to the user.

4.5. Discussion

4.5.1. ***Multi-R*** **Evaluation.** As we mentioned earlier, *Multi-R* was evaluated using simulated source statistics. Our evaluation can accurately measure the planning time of different algorithms and compare the utility of the generated query plans using the utility function. Although we cannot empirically evaluate the correctness of our proposed utility function without using real data sources within a real data integration system, we believe our utility model will work in real scenarios for the following reasons. First we adapted the traditional cost model to compute the response time which has been evaluated by the distributed DBMS community. Second the coverage model of a query plan is deduced directly from the plan coverage definition. It would be better if we can evaluate *Multi-R* in a real data integration system and directly use the coverage and overlap statistics learned by *StatMiner*. This evaluation was not done in the dissertation because of the difficulty in finding and/or developing a real data integration system which has a large number of mediated relations, as well as multiple integrated sources for each relation. Additionally since the scope of this dissertation covers mining and using coverage and overlap statistics, we did not discuss how other types of statistics (e.g. response time) are learned. In order to evaluate *Multi-R* in a real system, we would need techniques to gather other types of statistics, as the overall performance of the query optimizer will be dependent on all the statistics.

4.5.2. Multi-objective Query Optimization. The multi-objective problem is to find the best possible solution for problems in which there are several possibly opposing criteria or objectives. An optimum design problem must then be solved, with multiple objectives and constraints taken into account. This type of problem is known as either a multi-objective, multi-criteria, or a vector optimization problem [Azarm96].

In order to solve multi-objective problems, there are three popular approaches [DCD99]. One popular approach is to cast them into the conventional search framework after combining the multiple criteria into a single scalar criterion. However the task of constructing the combined evaluation function is context and user dependent[DCD99]. Another approach is to optimize one criterion at a time under given constraints on the others. The problem of the this approach is that we have to get a set of good constraints, in the absence of which search becomes unduly expensive. Moreover, repeatedly searching the same search space by progressively refining the constraints increase the search complexity tremendously [DCD99]. The third approach is the multi-objective search approach. It determines the set of all *undominated* solutions which is called a *Pareto curver* which captures the informational concept of a trade-off [DCD99].

In the query optimization scenario, since the users may have multiple objective w.r.t. query plans, we are interested in multi-objective problems too. Most of the existing approaches [SAP96, NLF99, NK01, DH02] use the first approach which combines the multiple criteria (such as cost, delay, coverage, quality) into a single scalar criterion (see Figure 39). The second approach is not used here because of its expensive planning cost and the difficulty of getting good constraints. The multi-objective search approach is rarely used because there are often exponentially many solutions on a Pareto curve. If we send all these solution plans to users, they may have to spend a lot time to decide which one is better (see Figure 40). Let me explain this using an example.

Example 1: Consider a *Simple Mediator* that integrates autonomous Internet sources. Let there be a single relation, R_1, in the global schema of the mediator. There are three sources: S_1, S_2 and S_3, each exporting a subset of the global relation. Let's assume the users have

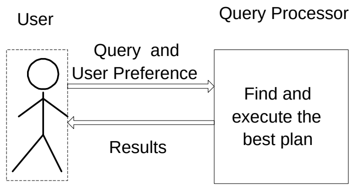


Figure 39. Query processing using combining multiple criteria approach

different interests in the coverage (number of answers) and the cost (response time) of the query plans. So for a simple selection query on R_1, there are 7 feasible query plans:

$P_1 = S_1$: $coverage = 0.5$ $cost = 6(seconds)$,

$P_2 = S_2$: $coverage = 0.4$ $cost = 7$,

$P_3 = S_3$: $coverage = 0.3$ $cost = 2$,

$P_4 = S_1 \cup S_2$: $coverage = 0.6$ $cost = 8$,

$P_5 = S_1 \cup S_3$: $coverage = 0.55$ $cost = 6.5$,

$P_6 = S_2 \cup S_3$: $coverage = 0.65$ $cost = 8.5$,

$P_7 = S_1 \cup S_2 \cup S_3$: $coverage = 0.8$ $cost = 13$

As we can see there is only one plan P_2 that is dominated by other plans. So there are still 6 undominated plans. If we send all the plans to the user, it may be difficult for them to make a decision in a short period of time. Here we just give a very simple example, however for more complicated queries, there may be hundreds of undominated plans. In order to pick the best plan, the user has to spend a lot of time to look at the plans and then make a decision. Since in many mediators, executing a plan may only take seconds, it will be prohibitively expensive to let user spend several minutes to pick a plan.

The paper [PY01] discussed how to solve the delay-cost tradeoff problem in the Mariposa database system [SAP96] by using the recent Pareto curve computing techniques to obtain in polynomial time an approximate trade-off (ϵ-Pareto curve), which is arbitrarily

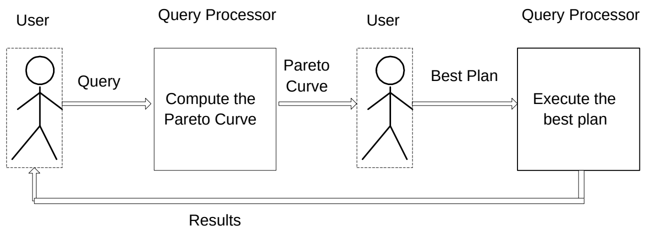


Figure 40. Query processing using pareto curve

close to the true Pareto curve. In this way they can trade the optimality of the solutions for speed. However the number of points (solutions) in ϵ-Pareto curve may still be large. This approach may be a good approach for problems with a small number of undominated solutions, or scenarios where the users are willing to spend long time on picking a close-to-optimum plan according to their preference. However in many other scenarios it will be better for the mediator to get the users' preference when (or before) the users submit their queries, make a decision automatically, and only show the final results to the user.

4.6. Summary

In this chapter we started by motivating the need for joint optimization of cost and coverage of query plans in data integration. We then argued that our way of searching in the space of parallel query plans, using cost models that combine execution cost and the coverage of the candidate plans, provides a promising approach. We described ways in which cost and coverage of a parallel query plan can be estimated, and combined into an aggregate utility measure. We then presented two algorithms to generate parallel query plans. The first, *ParPlan-DP*, is a System-R style dynamic programming algorithm, while the second, *ParPlan-Greedy*, is a greedy algorithm. Our experimental evaluation of these algorithms demonstrates that for a given coverage requirement, the plans generated by *Multi-R* are significantly better, both in terms of planning cost and in terms of the quality of the plans produced (measured in terms of its coverage and execution cost), compared to the existing approaches that use phased optimization using linear plans. We also demonstrated the flexibility of our algorithms in handling a spectrum of cost-coverage tradeoffs.

CHAPTER 5

RELATED WORK

5.1. Mining Coverage and Overlap Statistics

The utility of quantitative coverage statistics to rank the sources was first explored by Florescu *et. al.* [FKL97]. However, the primary aim of the effort was on modelling coverage and overlap statistics, and it did not discuss how such coverage statistics could be learned. The work requires a topic hierarchy, the manual creation of topic hierarchies is laborious and error-prone. In contrast, this dissertation provides a framework for automatically *learning* the required statistics without manual intervention.

There has been some previous work on learning database statistics both in multi-database literature and data integration literature. Much of it, however, focused on learning response time statistics. Zhu and Larson [ZL96] describe techniques for developing regression cost models for multi-database systems by selective querying. Adali *et. al.* [ACPS96] discuss how keeping track of rudimentary access statistics can help in doing cost-based optimizations. More recently, the work by Gruser *et. al.* [GRZ$^+$00] considers mining response time statistics for sources in a data integration scenario. In contrast, our work focuses on learning coverage and overlap statistics. As has been argued by us [NK01] and others [DH02], query optimization in data integration scenarios require both types of statistics.

Another strand of related work [WMY00, IGS01, IG02] considers the problem of text database selection in the context of keyword queries submitted to meta-search engines. Although some of these efforts use a hierarchy of topics to categorize the Web sources, they use only a single topic hierarchy and do not deal with computation of overlap statistics. In contrast we deal with classes made up from the cartesian product of multiple attribute value hierarchies, and are also interested in modeling overlap. This makes the issue of space consumed by the statistics quite critical for us, necessitating our threshold-based approaches for controlling the resolution of the statistics. Furthermore, most of the existing approaches in text database selection assume that the terms in a user's query are independent (to avoid storing too many statistics). No efficient approaches have been proposed to handle correlated keyword sets. We are currently working on applying our techniques to the text database selection problem to effectively solve the space and learning overhead brought by providing coverage and overlap statistics for both single word and correlated multi-word terms.

5.2. Multi-Objective Query Optimization

The Bucket algorithm [LRO96] and the source inversion algorithm [DGL00] provide two of the early approaches for generating candidate query plans in data integration. As we mentioned in Section 4.1 the disadvantages of generating all possible linear plans and executing them have lead to other alternate approaches. The work on Streamer project [DH02] extends the query planning algorithm in [LRO96], so it uses the coverage information to decide the order in which the potential plans are executed. The coverage is computed using the source overlap models in [FKL97]. A recent extension of [LRO96] is the MINICON algorithm presented in [PL00]. Although MINICON improves the efficiency of the bucket algorithm, it still assumes a decoupled strategy – concentrating on enumerating linear plans first, assessing their quality and executing them in a rank-ordered fashion next. The work by Naumann *et. al.* [NLF99] offers another variation on the bucket algorithm of [LRO96], where the set of linear plans are ranked according to a set of quality criteria, and a branch and bound approach is used to develop top-N best linear plans. Although their notion of quality seems to include both cost and coverage, their cost model seems to be quite restrictive, making their approach a phased one in essence. For example, they claim that "a change of the join execution order within a plan has no effect on its IQ [quality] score." As we have seen, join orders do have a significant impact on the overall quality (utility) of the plan.

Although [YLUG99] and [FLMS99] consider the cost-based query optimization problem in the presence of binding patterns, they do not consider the source selection issue in their work. Finally, parallel plans (or joins over unions) are quite standard in distributed/parallel database systems [LPR98; OV99]. Use of parallel plans in data-integration scenarios does however pose several special challenges because of the uncontrolled overlap between data sources, the source access (binding) restrictions, and the need to produce plans for a variety of cost/coverage requirements.

5.3. Relation to Adaptive Query Optimization

Recently several adaptive query evaluation approaches including Tukwila [IFF$^+$99], Query Scrambling [UFA98, UF00], Eddies [AH00, MSHR02] are developed to handle the problem of lack of statistics from autonomous online sources, and the unanticipated delays and failures of the online sources. Most of them interweave optimization during execution.

Here we briefly describe these approaches, and show how my dissertation work is connected to their approaches.

5.3.1. Existing Adaptive Query Evaluation Approaches.

5.3.1.1. *Tukwila.* Tukwila is developed to provide query evaluation techniques in data integration scenarios where few statistics are collected from the autonomous sources, and unanticipated delays and failures of the online sources are expected. Tukwila interleaves optimization and execution at the core, which allows it to recover from decisions based on inaccurate estimates. For a given query, the query optimizer produces query execution plans via a System-R style dynamic programming algorithm. However the optimizer does not always create a complex execution plan for the query. If essential statistics are missing or uncertain, the optimizer may generate a partial plan, deferring subsequent planning until sources have been contacted and critical metadata obtained. During execution, Tukwila also checks at materialization points in order to detect opportunities for re-optimization. Tukwila uses query operators that are especially suited for adaptive behavior - the double pipelined hash join, which produces answers quickly, and the dynamic collector, which robustly and efficiently computes unions across overlapping data sources.

5.3.1.2. *Query Scrambling.* In [UFA98], query scrambling has been introduced to modify query execution plans on-the-fly when delays are encountered during runtime. In the paper, the authors addressed the shortcoming of the heuristic-based approach and proposed three different cost-based approaches to using query optimization for scrambling. In the experiments, the authors use a two-phase randomized optimizer to explore the search space. XJoin [UF00] introduced a reactively scheduled pipelined join operator, which is similar to the double pipelined join operator (DPHJ). Compared to Xjoin, however, DPHJ does not include reactively-scheduled background processing for coping with delayed sources.

5.3.1.3. *Eddies.* In [AH00], the authors introduced a query processing mechanism called an eddy, which continuously reorders operators in a query plan as it runs. By combining eddies with appropriate join algorithms, eddies can merge the optimization and execution phase of query processing, allowing each tuple to have a flexible ordering of query operators. A heuristic pre-optimizer is used to choose how to initially pair off relations into joins.

5.3.2. Parameterized Decision Model. From the above discussion, we can see that actually both Tukwila and Query Scrambling use a parameterized decision model and explore the search space (Tukwila uses a System-R style dynamic programming algorithm, Query Scrambling uses a two-phase randomized optimizer) to find the best initial plan, and re-optimize a bad performance (or a partial) plan. Eddies do not use any parameterized decision model and explore a search space. The reasons why eddies do not use traditional static optimization techniques are that:

- In wide-area environments, the resources in a distributed environment can exhibit widely fluctuating characteristics, and their performance may be unpredictable;
- Selectivity estimation is often quite inaccurate;
- In large-scale systems, many queries can run for a very long time.

As a result, assumptions made at the time a query is submitted will rarely hold throughout the duration of query processing.

If a query optimizer does not use any parameterized decision model and explore a search space, it will face the following issues:

1. Although a reactive processor can run multiple alternative plans (or subplans) at the beginning, and adaptively choose among them over time. However it may be too costly to execute all the feasible plans at the beginning stages of evaluating a query. This is because we may have a lot of feasible plans available, especially in the data integration scenario where for each mediated relation we may have dozens or hundreds of source relations export information for it. In order to avoid wasting too many computing resources on unpromising alternatives, the optimizer has to select a small set of good plans. A heuristic based pre-optimization approach can be used to find these initial plans. However, as the previous work shows, the utility of the query plan generated by heuristic based approaches can be arbitrarily far away from optimal;

2. Changes of some query plans on the fly may require significant processing and code complexity to guarantee correct results. This is because many join operators (and other operators) cannot be easily reordered;

3. Eddies assume that the queries take a very long time to run, and during the running of the plan, the environment changes dramatically. However, in many data integration scenarios, this may not be the case. More importantly, other parameters of the environment (such as coverage and overlap of the sources and number of answer tuples of a query from a source) have a very low chance of changing dramatically during the running of the plan.

5.3.3. Adaptive Query Processing and my Dissertation Work. My research shares the same motivation of the above approaches: the sources are autonomous, so very few statistics can be expected from the sources. We are trying to solve the lack of statistics by automatically learning the (coverage) statistics from the sources for a mediator using a novel association rule mining approach, and adaptively updating the statistics as the environment changes. With these learned up-to-date statistics the query engines (including Tukwila and Query Scrambling) can generate more accurate initial query plans, which will definitely improve the performance of these query engines by avoiding too many unnecessary reoptimization.

We also provide a joint optimization approach to select plans (or initial plans) based on both the response time and coverage of the plans. The existing adaptive query evaluation approaches can be view as complementary to my dissertation work, since my dissertation does not focus on plan execution, while Tukwila and Query Scrambling do not focus on the query joint optimization. The replanning approach in Tukwila and Query Scrambling could be used to handle the unexpected delays of the source calls in the initial plans generated by our multi-objective approach. As the authors of Eddies mentioned, eddies can be viewed as complementary work to Tukwila and Query Scrambling. Eddies can be used to do tuple scheduling within pipelines, and Tukwila and Query Scrambling can be used to reoptimize across pipelines.

CHAPTER 6

CONCLUSION and FUTURE DIRECTIONS

The dissertation motivated the need for automatically learning the coverage and overlap statistics of Internet sources and the need for joint optimization of cost and coverage of parallel query plans for efficient query processing in a data integration scenario. We have proposed a set of connected techniques for efficient query processing:

StatMiner: Automated learning of Attribute Value hierarchies, and using threshold based data mining techniques to discover frequent query classes (or large classes in scenarios without query distributions) and to learn their coverage and overlap statistics;

Multi-R: Searching in the space of parallel query plans, and using cost models that combine execution cost and coverage of the candidate plans.

As part of my dissertation work, *StatMiner* has been implemented and evaluated in the context of *BibFinder*, and *Multi-R* has been implemented and evaluated in a synthetic dataset. The empirical evaluation shows that:

- *StatMiner* can systematically trade the statistics learning time and memory space consumption of statistics remembered for accuracy by varying the frequent class thresholds;

- The learned statistics by *StatMiner* provide tangible improvements in the source ranking, and the improvement is proportional to the type (coverage alone vs. coverage and overlap) and granularity of the learned statistics;
- The plans generated by Multi-R were significantly better compared to the existing approaches, both in terms of planning cost, and in terms of plan execution cost.

I believe my dissertation work has made significant advances towards solving the problem of source selection and multi-objective query optimization in data integration systems. However, there are several important directions in which my dissertation work could be extended.

6.1. Incremental Update of Statistics

One of the simplifying assumptions in our source statistics mining work to date is that queries asked by the users in future will have the same distribution as the past queries. Since the users' interests may change over time, an important extension is to incrementally update the learned statistics with respect to the users' most recent interests. To handle shifting interests of the user population, we propose to keep track of a sliding window of user queries and incrementally recompute the "frequent" query classes. This in turn involves updating the AV-hierarchies (see Section 3.4).

Because of the dynamic nature of the sources, the coverage and overlap of the sources could change over time. In order to dynamically update the statistics with respect to the changes of the integrated sources, we propose to periodically probe all sources using the user queries and check the inconsistency between the estimated statistics and the current real coverage and overlap of the sources. If a large inconsistency is found, *StatMiner* re-learns the statistics with respect to the corresponding query class by probing the sources using the queries subsumed by the class.

6.2. First Tuples Fast

Traditional query optimization focused on optimizing the response time [GHK92] of the last tuple of a plan or the total work [SACL79] of executing a plan. However if the users are interested in getting first answer tuples fast, which becomes more and more important in the Internet data integration scenario, then even if two plans have the same total response time (or total work) and the same number of answers, the utility of those two plans may still be very different because the response time for their first k answers may be very different [VN02]. In order to evaluate how good the performance of these plans are in terms of first tuples fast, we propose a utility model which takes the number of tuples, the response time of these tuples, and the total work into account:

$$utility(p) = reward(p) + \beta * work(p)$$

$$reward(p) = \sum_i \gamma^{t_i} numoftuple_i$$

Here β is an adjustable parameter which may depend on the workload of the system, t_i is the ith stage of the plan execution period, $numoftuple_i$ is the number of tuples we get at stage t_i, and γ^{t_i} is an adjustable discount factor for the tuples delivered at stage t_i. we believe including this reward function with delay penalty in *Multi-R* would lead to plans giving first answer tuples fast.

6.3. Applications

There are opportunities to apply some of the lessons from my research to other domains. One direction is the integration of Bioinformatic data sources. There has been an increased interest in approaches for effectively integrating the hundreds of public-domain biological sources containing data on classes of scientific entities such as genes and sequences. Each source may have data on one or more classes. There is significant diversity in the coverage of these sources. Relationships between scientific objects are often implemented as physical links between data sources [LNRV03]. Since biologists are often interested in exploring relationships between scientific objects stored in different sources, our work in multi-objective query optimization can be extended to automatically select efficient query plans (paths) from the many alternative paths, and my work in source coverage and overlap statistics learning can be extended to mine path coverage and overlap statistics.

Another direction is database selection in distributed information retrieval. My work in mining coverage and overlap statistics can be extended to select text databases in distributed information retrieval. In order to avoid storing too many statistics, most of the existing approaches to select relevant text databases in distributed information retrieval assume that the terms in a user's query are independent and that sources are not overlapping. Our frequency-based coverage and overlap mining approach could effectively solve the space and learning overhead brought by providing coverage and overlap statistics for both single word and correlated multi-word terms.

REFERENCES

[AH00] R. Avnur and J. Hellerstein. Eddies: Continuously Adaptive Query Processing. In *Proceedings of SIGMOD*, 2000.

[AM] Amazon. http://www.amazon.com.

[AS94] Rakesh Agrawal, Ramakrishnan Srikant. Fast Algorithms for Mining Association Rules. In *VLDB*, Santiage, Chile, 1994.

[Azarm96] Shapour Azarm. Multiobjective Optimum Design. 1996.
http : //www.glue.umd.edu/ azarm/optimum_notes/multi/multi_TOC.html.

[BF] BibFinder. *http : //rakaposhi.eas.asu.edu/bibfinder.*

[BFG01] Robert Baumgartner, Sergio Flesca and Georg Gottlob. Visual Web Information Extraction with Lixto. In *VLDB conference*, 2001.

[C01] K.S. Candan. Query optimization in Multi-media and Web Databases. ASU CSE TR 01-003. Computer Science & Engg. Arizona State University.

[C77] William G. Cochran. Sampling Techniques. John Wiley & Sons. Third edition, 1977.

[CFN77] G. Cornuejols, M. L. Fisher and G. L. Nemhauser. Locations of bank accounts to optimize float: an analytic study exact and approximate algorithms. Management Science. 23 (1977) 789-810.

[CGMH94] Sudarshan Chawathe, Hector Garcia-Molina, Joachim Hammer, Kelly Ireland, Yannis Papakonstantinou, Jeffrey Ullman, and Jennifer Widom. The TSIMMIS project: Integration of heterogeneous information sources. In *Proceedings of the 100th Anniversary Meeting*, pages 7–18, Tokyo, Japan, October 1994. Information Processing Society of Japan.

[CMM01] Valter Crescenzi, Giansalvatore Mecca, and Paolo Merialdo. Roadrunner: Towards automatic data extraction from large web sites. In *VLDB conference*, 2001.

[DCD99] P. Dasgupta,P. Chakrabarti and S. DeSarkar. Multiobjective Heuristic Search. Vieweg, Bertelsmann Professional International, Germany, 1999 (ISBN 3-52805-708-4).

[DG97] Oliver M. Duschka and Michael R. Genesereth. Answering recursive queries using views. In *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, PODS '97*, pages 109 – 116, Tucson, AZ, May 1997.

[DGL00] Oliver M. Duschka, Michael R. Genesereth, Alon Y. Levy. Recursive Query Plans for Data Integration. In *Journal of Logic Programming, Volume 43(1)*, pages 49-73, 2000.

[DH02] A. Doan and A. HaLevy. Efficiently Ordering Plans for Data Integration. In *Proceedings of ICDE-2002*, 2002.

[DL97] Oliver M. Duschka and Alon Y. Levy. Recursive plans for information gathering. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence, IJCAI*, Nagoya, Japan, August 1997.

[F96] U. Feige. A threshold of ln(n) for approximating set cover. In *Proceeding of the 28th annual ACM Symposium on the Theory of Computing*, pp. 314-318, 1996.

[FKL97] D. Florescu, D. Koller, and A. Levy. Using probabilistic information in data integration. In *Proceeding of the International Conference on Very Large Data Bases* (VLDB), 1997.

[FLMS99] D. Florescu, A. Levy, I. Manolescu, and D. Suciu. Query optimization in the presence of limited access patterns. In *Proc. SIGMOD*, 1999.

[FW97] Marc Friedman and Daniel S. Weld. Efficiently executing information-gathering plans. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence, IJCAI*, Nagoya, Japan, August 1997.

[GGKS95] Donald F. Geddis, Michael R. Genesereth, Arthur M. Keller, and Narinder P. Singh. Infomaster: A virtual information system. In *Intelligent Information Agents Workshop at CIKM '95*, Baltimore, MD, December 1995.

[GHK92] W. Ganguly, S. Hasan and R. Krishnamurthy. Query Optimization for Parallel Execution. In *Proceedings of SIGMOD*, 1992.

[GRZ$^+$00] Jean-Robert Gruser, Louiqa Raschid, Vladimir Zadorozhny, Tao Zhan: Learning Response Time for WebSources Using Query Feedback and Application in Query Optimization. VLDB Journal 9(1): 18-37 (2000)

[H97] D. S. Hochbaum. Approximation algorithms for NP-hard problems. PWS Publishing Company, 1997.

[HC01] Theodore W. Hong and Keith L. Clark. Using Grammatical Inference to Automate Information Extraction from the Web. In Principles of Data Mining and Knowledge Discovery, Lecture Notes in Computer Science, volume 2168, pages 216-227. Springer-Verlag, 2001.

[HD98] Chun-Nan Hsu and Ming-Tzung Dung. Generating Finite-State Transducers for Semi-Structured Data Extraction from the Web. Information Systems, 23(8):521-538, 1998.

[HK00] Jiawei Han and Micheline Kamber. Data Mining: Concepts and Techniques. Morgan Kaufmman Publishers, 2000.

[HKWY97] Laura M. Haas, Donald Kossmann, Edward L. Wimmers, and Jun Yang. Optimizing queries across diverse data sources. In *Proc. VLDB*, 1997.

[IFF$^+$99] Z. Ives, D. Florescu, M. Friedman, A. Levy and D. Weld. An Adaptive Query Execution System for Data Integration. In *Proceedings of SIGMOD*, 1999.

[IG02] P. Ipeirotis and L. Gravano. Distributed Search over the Hidden-Web: Hierarchical Database Sampling and Selection. In *Proc. of VLDB*, 2002.

[IGS01] P. Ipeirotis, L. Gravano, M. Sahami. Probe, Count, and Classify: Categorizing Hidden Web Dababases. In *Proceedings of SIGMOD-01*, 2001.

[KLN$^+$04] Subbarao Kambhampati, Eric Lambrecht, Ullas Nambiar, Zaiqing Nie, and Gnanaprakasam Senthil. Optimizing Recursive Information Gathering Plans in EMERAC. *Journal of Intelligent Information Systems*, Volume 22, Issue 2 (March 2004), Pages: 119 - 153, 1999.

[Kus00] Nicholas Kushmerick. Wrapper Verification. *World Wide Web*, 3(2):79-94, 2000.

[KW96] C. Kwok, and D. Weld. Planning to gather information. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, 1996.

[KWD97] Nicholas Kushmerick, Daniel S. Weld, and Robert Doorenbos. Wrapper Induction for Information Extraction. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, 1997.

[Lev00] A. Levy. Theory of answering queries using views. *SIGMOD Record*, 2000.

[LK97] Eric Lambrecht and Subbarao Kambhampati. Planning for information gathering: A tutorial survey. Technical Report ASU CSE TR 97-017, Arizona State University, 1997. rakaposhi.eas.asu.edu/ig-tr.ps.

[LKG99] E. Lambrecht, S. Kambhampati and S. Gnanaprakasam. Optimizing recursive information gathering plans. In *Proceeding of the International Joint Conference on Artificial Intelligence (IJCAI)*, 1999.

[LKM01] Kristina Lerman, Craig A. Knoblock and Steven Minton. Automatic Data Extraction from Lists and Tables in Web Sources. In *Proceedings of the IJCAI 2001 Workshop on Adaptive Text Extraction and Mining*, Seattle, WA, 2001.

[LMK03] Kristina Lerman, Steven Minton and Craig Knoblock . Wrapper Maintenance: A Machine Learning Approach. *Journal of Artificial Intelligence Research*, 18:149-181, 2003.

[LNRV03] Zoe Lacroix, Felix Naumann, Louiqa Raschid, and Maria Esther Vidal. Exploring Life Sciences Data Sources. In *Proceedings of IJCAI'03 Workshop on Information Integration on the Web*, August 9-10, 2003, Acapulco, Mexico.

[Lowell03] The lowell database research self assessment. June 2003. http://research.microsoft.com/∼ *gray/lowell*

[LPR98] Ling Liu, Calton Pu, Kirill Richine. Distributed Query Scheduling Service: An architecture and its Implementation. In *Special issue on Compound Information Services, International Journal of Cooperative Information Systems (IJCIS).* Vol.7, No.2&3, 1998. pp123-166. 1998.

[LRO96] A. Levy, A. Rajaraman, J. Ordille. Query Heterogeneous Information Sources Using Source Descriptions. In *VLDB Conference*, 1996.

[MMK01] Ion Muslea, Steve Minton, and Craig Knoblock. Hierarchical Wrapper Induction for Semistructured Information Sources. *Journal of Autonomous Agents and Multi-Agent Systems*, 4:93-114, 2001 .

[MSHR02] S. Madden, M. Shah, J. Hellerstein and V. Raman. Continuously Adaptive Continuous Queries over Streams. In *Proceedings of SIGMOD*, 2002.

[Nau02] F. Naumann. Quality-Driven Query Answering for Integrated Information Systems. Volume 2261 of LNCS, Springer Verlag, Heidelberg, 2002.

[NLF99] F. Naumann, U. Leser, J. Freytag. Quality-driven Integration of Heterogeneous Information Systems. In *VLDB Conference* 1999.

[NK01] Z. Nie and S. Kambhampati. Joint optimization of cost and coverage of query plans in data integration. In *Proceedings of CIKM*, Atlanta, Georgia, November 2001.

[NK04] Z. Nie and S. Kambhampati. A Frequency-based Approach for Mining Coverage Statistics in Data Integration. In *Proceedings of ICDE*, 2004.

[NKH03] Z. Nie, S. Kambhampati and T. Hernandez. BibFinder/StatMiner: Effectively Mining and Using Coverage and Overlap Statistics in Data Integration. In *Proceedings of VLDB*, 2003.

[NNVK02] Z. Nie, U. Nambiar, S. Vaddi and S. Kambhampati. Mining Coverage Statistics for Websource Selection in a Mediator. Proc. CIKM 2002.

[OV99] M.T. Ozsu and P. Valduriez. Principles of Distributed Database Systems (2nd Ed). Prentice Hall. 1999.

[PL00] Rachel Pottinger , Alon Y. Levy , A Scalable Algorithm for Answering Queries Using Views Proc. of the Int. Conf. on Very Large Data Bases(VLDB) 2000.

[PY01] C. Papadimitriou and M. Yannakakis. Multiobjective Query Optimization. In *PODS* 2001.

[Qia96] Xiaolei Qian. Query folding. In *Proceedings of the 12th International Conference on Data Engineering*, pages 48–55, New Orleans, LA, February 1996.

[SACL79] P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, T. Price. Access path selection in a relational database management system. In *SIGMOD* 1979.

[SAP96] M. Stonebraker, P. M. Aoki, A. Pfeffer, A. Sah, J. Sidell, C. Staelin, and A. Yu. Mariposa: A Wide-Area Distributed Database System. VLDB journal, 5:1, pp. 48-63, 1996.

[TPC] Transaction Processing Council. http://www.tpc.org.

[TRV98] A. Tomasic, L. Raschid, and P. Valduriez. Scaling access to heterogeneous data sources with disco. *IEEE TKDE*, 10(5), 1998.

[UF00] XJoin: Tolga Urhan, Michael J. Franklin. A Reactively Scheduled Pipelined Operator. In *IEEE Data Engineering Bulletin*, 23(2), 2000.

[UFA98] Tolga Urhan, Michael J. Franklin, and Laurent Amsaleg. Cost-Based Query Scrambling for Initial Delays. In *Proceedings of SIGMOD*, Seattle, WA, June, 1998.

[VN02] Efstratios Viglas and Jeffrey F. Naughton. Rate-based Qurey Optimization for Streaming Information Sources. In *Proceedings of SIGMOD*, 2002.

[WMY00] W. Wang, W. Meng, and C. Yu. Concept Hierarchy based text database categorization in a metasearch engine environment. In *Proceedings of WISE*, June 2000.

[YLUG99] R. Yerneni, C. Li, J. Ullman and H. Garcia-Molina. Optimizing large join queries in mediation systems. In *Proc. International Conference on Database Theory*, 1999.

[Zipf29] George Kingsley Zipf. Relative Frequency as a Determinant of Phonetic Change. Harvard Studies in Classical Philology, Vol. 40. pp. 1-95, 1929.

[ZL96] Q. Zhu and P-A. Larson. Developing Regression Cost Models for Multi-database Systems. In *Proceedings of PDIS*, 1996.

[ZRV$^+$02] Vladimir Zadorozhny, Louiqa Raschid, Maria-Esther Vidal, Tolga Urhan, Laura Bright. Efficient evaluation of queries in a mediator for WebSources. In *Proceedings of SIGMOD*, 2002.

BIOGRAPHICAL SKETCH

Zaiqing Nie will be joining the Information Management and System Group at the Microsoft Research Asia as an Associate Researcher in May 2004. His research interests include data integration, data mining, information retrieval, and data stream management.

He will be graduating in May 2004 with a Ph.D. in Computer Science from Arizona State University. He received his Master of Engineering degree in Computer Applications from Tsinghua University in 1998, and his Bachelor of Engineering degree in Computer Science and Technology from Tsinghua University in 1996.