ASU CSE Technical Report 05-003
May, 2005

Mining, Using and Maintaining Source Statistics for Adaptive Data Integration

Jianchun Fan and Subbarao Kambhampati
Arizona State University
and
Zaiqing Nie
Microsoft Research Asia

To make query processing effective in data integration scenarios, the mediator needs to be able to gather and use statistics about data sources as well as to adapt to the often conflicting user preferences. We present a framework for effectively mining multiple types of statistics including source coverage statistics, inter-source overlap statistics and source latency profiles. Using these statistics enables the mediator to optimize the coverage and execution cost respectively. However, users in data integration systems often require query plans that are optimal with respect to multiple objectives and such objectives often conflict. We present a joint optimization model that uses latency as well as coverage/overlap statistics simultaneously to support a spectrum of trade-offs between the coverage and latency requirements of query plans. Moreover, motivated by the dynamic and evolving nature of data integration systems, we introduce an incremental approach for maintaining source statistics.

We describe the details of our approaches and present extensive experimental results in the context of *Bibfinder*, a fielded bibliographic mediator system. Our results demonstrate the effectiveness of statistics learning and maintenance and multi-objective optimization.

Categories and Subject Descriptors: H.2.5 [**Database Management**]: Heterogeneous Databases; H.2.4 [**Database Management**]: Systems—*Query Processing*; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval; H.3.5 [**Information Storage and Retrieval**]: Online Information Services

General Terms: Algorithms, Design, Experimentation, Performance

Additional Key Words and Phrases: Data Integration, Multi-Objective Optimization, Statistics Mining, Incremental Maintenance

1. INTRODUCTION

The availability of structured information sources on the web has recently led to significant interest in query processing frameworks that can integrate information sources available on the Internet. Data integration systems [Duschka et al. 2000;

Parts of this paper–dealing with the learning of coverage and overlap statistics–have been presented at ICDE 2004 [Nie and Kambhampati 2004]. This paper significantly expands the ICDE paper by (1) Considering incremental maintenance of statistics (2) learning source latency statistics and (3) considering multi-objective source selection using coverage as well as latency statistics. This paper is also differs in content and approach from a related journal publication in IEEE TKDE [Nie et al. 2004]. Please see the related work section for details.

Authors' Addresses: (Authors' names listed in alphabetical order) Jianchun Fan and Subbarao Kambhampati are with the Department of Computer Science and Engineering, Arizona State University, Tempe, AZ 85287-5406, email: {jcf, rao}@asu.edu; Zaiqing Nie is with Microsoft Research Asia, 5F Beijing Sigma Center, No.49 Zhichun Road, Haidian District, Beijing, PR China, 100080, email: t-znie@microsoft.com.

Adali et al. 1996; Lambrecht et al. 1999; Levy et al. 1996; Nie et al. 2003; Pottinger and Levy 2000] 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. Not only must a data integration system adapt to the variety of sources available, but it also has to take into account multiple and possibly conflicting user objectives. Such objectives can include overall coverage objectives, cost-related objectives (e.g. response time [Gruser et al. 2000]), and data quality objectives (e.g. density [Naumann et al. 2004], provenance of results [Buneman et al. 2001]). To address these conflicting user preferences, query optimization in data integration requires the ability to figure out what sources are most relevant both to the given query and to the specific user objectives. For this purpose, the query optimizer needs to access several types of source-specific statistics and perform a multi-objective optimization of the query using these statistics.

Unfortunately, the autonomous and decentralized nature of the data sources constrains the mediators to operate with very little information about the structure, scope, contents, and access costs of the information sources they are trying to integrate. While some types of statistics may well be voluntarily publicized by the individual data sources, many of these statistics need to be learned/gathered, and actively maintained by the mediator. This thus raises three challenges:

(1) How to automatically gather various statistics from autonomous sources,
(2) How to use the gathered statistics to support multi-objective query processing, and
(3) How to maintain the gathered statistics to keep up with the continuously evolving system.

Not surprisingly, due to the lack of available source statistics, most existing integration frameworks are unable to support flexible query processing that takes conflicting user preferences into account.

Our work is motivated and evaluated in the context of *Bibfinder* (Figure 1, http://rakaposhi.eas.asu.edu/bibfinder), a fielded bibliography mediation system. *Bibfinder* currently integrates several computer science bibliography sources including *ACM Digital Library*, *ACM Guide*, *Network Bibliography*, *IEEE Explorer*, *DBLP*, *CSB* and *Sciencedirect*. The global schema of *Bibfinder* is a single relation: $Paper(title, author, conference/journal, year)$. Each source only covers a subset of the global schema. *Bibfinder* presents a unified and more complete view to the users to query these bibliography sources. The queries submitted to *Bibfinder* are selection queries on the *Paper* relation and they are directed to the most relevant data sources, and the tuples returned by the sources are presented to the users. Since its unveiling in December 2002, *Bibfinder* has answered more than 100,000 queries which were logged and used in our statistics mining and evaluation.

BibFinder offers interesting contrasts with respect to other bibliography search engines like CiteSeer [CiteSeer]. First of all, because it uses online integration approach (rather than a data warehouse one), user queries are sent directly to the integrated Web sources and the results are integrated on the fly to answer a query. This obviates the need to store and maintain the paper information locally. Secondly, the sources integrated by *BibFinder* are autonomous and partially overlapping. By combining the sources, *BibFinder* can present a unified and more complete

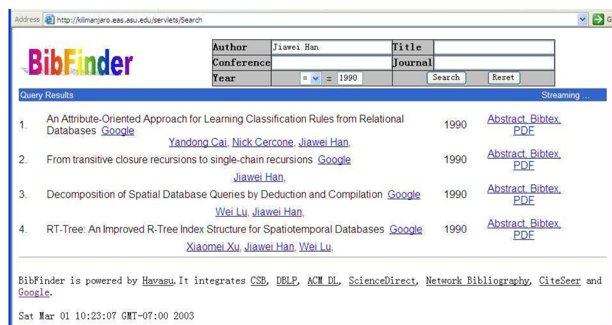


Fig. 1. The *BibFinder* User Interface

view to the user. These features require the *Bibfinder* to tackle the challenges of mining, using and maintaining various types of source statistics as discussed above. Specifically, the following are several motivating examples from this scenario:

—A naive way to answer a query is to send it to all the integrated sources, collect the answers from them and show it to the user. However, some sources may not have any answers for that query, and different sources may export same answers. To call every source for every query will unnecessarily increase the work load of the sources. In stead, the mediator should select only a relevant subset of the sources to be called for a given query. To make such source selection, the *Bibfinder* needs to know the degree of relevance of each source to that query (*coverage*), and the size of intersection of answer sets among multiple sources (*overlap*). These statistics are not directly available to the mediator. They have to be mined by the mediator and used in the query processing to figure out the most relevant subset of sources for a given query.

—For a given bibliography query, the user may want to have *some* bibliography entries found as quickly as possible and doesn't care about the *completeness* of the answer set. In this case, the mediator needs to know for that query, which source is able to respond with the shortest delay. Thus such statistics (*latency*) also needs to be learned by the mediator.

—A user might want a query plan to be optimal in multiple dimensions simultaneously. For example he/she might want the query to be answered as quickly as possible *and* as completely as possible. Therefore the mediator needs to combine the multiple types of statistics together and tries to jointly optimize the multiple objectives.

—The *Bibfinder* system is constantly evolving. New bibliography entries are inserted into the individual sources on a daily basis; the focus of interest of the user population shifts rapidly with the emergence of new "hot spots" of computer science literature; the sources may upgrade their server and network topology might change. All these changes indicate that the corresponding statistics (coverage/overlap and latency) learned by the mediator have to be updated accordingly so that they can keep being effectively used in query processing.

We have been developing an adaptive data integration framework to address

the challenges above. Figure 2 shows the architecture of this framework. Our contribution can be classified into 3 parts:

1. Mining Query Sensitive Statistics: In this framework the statistics mining module (*Statminer*) can automatically identify a set of "frequently accessed query classes" and gathers source coverage/overlap statistics with respect to them. *Statminer* also gathers source latency profiles in a query sensitive way by learning them with respect to the binding patterns of the queries.

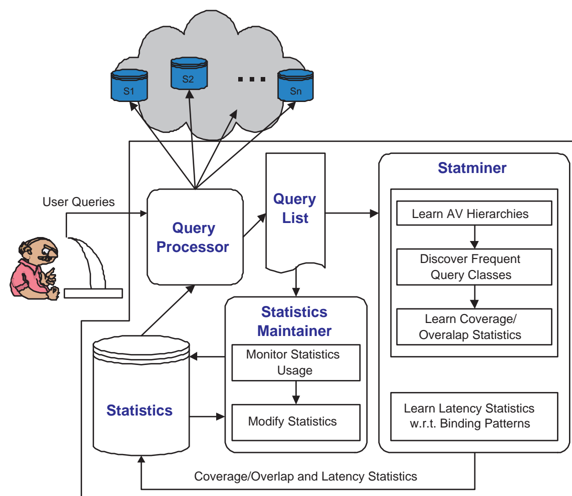


Fig. 2. The Adaptive Data Integration Architecture

2. Supporting Multi-Objective Query Processing: With the learned statistics the mediator will be able to make source selection to optimize individual objectives such as higher coverage or lower latency, using coverage/overlap and latency statistics respectively.

In data integration systems, the users usually prefer query plans that are optimal in terms of multiple objectives. Unfortunately such objectives often conflict. A plan optimal with respect to one objective cannot in general be post-processed to account for the other objectives. For example, sources with high coverage are not necessarily the ones that respond quickly. Therefore the mediator will need to generate a plan optimizing both coverage and latency so that the users can get most answers as soon as possible. The query processor in our framework adopts a novel joint optimization model in source selection to resolve conflicting user requirements with respect to coverage and latency and make flexible trade-offs between them, using both the coverage/overlap and latency statistics gathered.

3. Incremental Statistics Maintenance: Data integration systems usually evolve continuously over time. As a result, the statistics of the sources have to be updated accordingly to keep their accuracy. Maintenance of statistics is not always a trivial task of relearning, considering the impact of learning cost on the scalability

of the system. We introduce an incremental statistics maintenance approach for the purpose of reducing maintenance cost and keeping statistical accuracy.

We have evaluated the mining, usage and maintenance aspects of our adaptive data integration framework on *Bibfinder* test bed. We will demonstrate in Section 3.1.4 that the *Statminer* is able to learn source coverage, inter-source overlap and source latency statistics effectively. The learned statistics are able to estimate the statistics for new queries with high accuracy and thus enhance the source selection task of the query processor. We will then show in Section 3.2.2 that the joint optimization model we propose can effectively resolve confliction of coverage and latency and make flexible trade-offs between them. Finally we will show in Section 4.3 that our incremental statistics maintenance approach can significantly reduce the learning cost and keep the statistics accurate.

The rest of this paper is organized as follows. In the next section, we will first introduce *Statminer* which automatically discovers frequent query classes and learns source coverage/overlap statistics. Then we will show how the source latency statistics are learned in a query sensitive way. In Section 3 we will demonstrate how using learned statistics can help source selection. We will discuss the need of multi-objective query optimization, and present our joint optimization model and show the experimental evaluation on it. In Section 4 we will discuss the motivation and present the solution and empirical evaluation of incremental statistics maintenance approach. We will discuss related work in Section 5 and summarize our conclusion in Section 6.

2. MINING SOURCE STATISTICS

In *Bibfinder*, the mediator keeps track of a query list which contains the past user queries. For each query it keeps information of how often it is asked and how many answers came from each source and source set. Figure 3 shows a fragment of the query list in *Bibfinder*. The query list is used in learning source statistics.

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

Fig. 3. A fragment of query list in *Bibfinder*.

2.1 Mining Coverage and Overlap Statistics

To efficiently answer user queries, it is important to find and access the most relevant subset of the sources for the given query. Suppose, the user asks a selection query
Q(title,author,year) :−
  paper(title, author, conference/journal, year),
  conference/journal = "SIGMOD".
A naive way of answering this selection query would be to send it to all the data 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. For example, for the selection query above, *DBLP* and *ACM Digital Library* is most relevant, and *Network Bibliography* is much less relevant. Furthermore, since *DBLP* stores records of virtually all the SIGMOD papers, a call to *ACM Digital Library* is largely redundant[1].

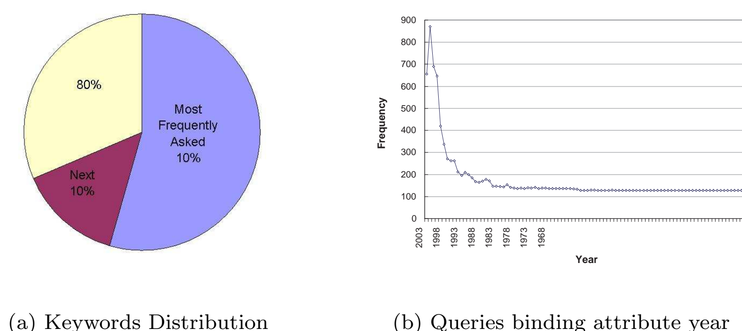
2.1.1 *Coverage and Overlap Statistics.* In order to judge the source relevance however, *Bibfinder* needs to know the *coverage* of each source S with respect to the query Q, i.e. $P(S|Q)$, the probability that a random answer tuple for query Q belongs to source S. Given this information, we can rank 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'.

If we have the coverage and overlap statistics for every possible query, we can get the complete order in which to access the sources. However it would be very costly to learn and store statistics w.r.t. 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. The second is the cost of "storing"the statistics.

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,

[1] 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 paper, all our references to *BibFinder* are to its first phase.

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. *Bibfinder* facilitates this by maintaining a log of queries, and for each query the statistics on how many of the query answers came from which sources. 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. 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 Zipfian. This belief is amply validated in *BibFinder*. Figure 4 shows the distribution of the keywords, and bindings for the Year attribute used in the first 15000 queries that were posed to *Bibfinder*. Figure 4(a) shows that the most frequently asked 10% keywords appear in almost 60% of all the selection queries binding attribute Title. Figure 4(b) shows that the users are much more interested in recently published papers.



(a) Keywords Distribution (b) Queries binding attribute year

Fig. 4. Query Distributions in BibFinder

2.1.2 *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.

The foregoing discussion about query classes raises the issue regarding the way query classes are defined to begin with. 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 5 shows an example of AV hierarchies and the corresponding query classes (see Section 2.1.3 for details). AV hierarchies could be hand-developed or automatically generated (see Section 2.1.4) using clustering techniques. 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]

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*. In this paper, we describe the details of the *Statminer* approach, and its use in *Bibfinder*. *Statminer* starts with a list of workload queries. The query list is collected from the logs 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 of each query came from which source. The query list is used to automatically learn AV hierarchies. The space of query classes is then defined in terms of the product of these AV hierarchies. The query classes are further pruned such that only those classes that subsume more than a given number of queries (specified by a frequency threshold) are retained. 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.

In the rest of this section we first define some terminology about query classes and AV hierarchies. Then we introduce the algorithms for learning AV hierarchies and discovering frequent query classes. We then describe how coverage and overlap statistics are learned for the frequent query classes, and how the learned statistics are used to develop a query plan for new queries.

2.1.3 *AV Hierarchies and Query Classes.* **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

[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!

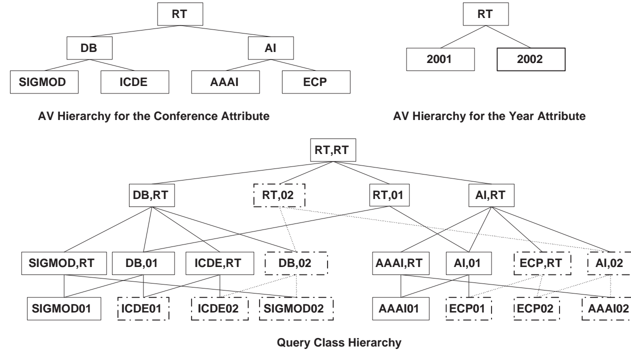


Fig. 5. AV Hierarchies and the Corresponding Query Class Hierarchy

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 5 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**. We can choose as the classificatory attributes the best k attributes whose values differentiate the sources the most, where the number k is decided based on a tradeoff between prediction performance and the computational complexity of learning the statistics by using these k attributes. The selection of the classificatory attributes may either be done by the mediator designer or using automated techniques. Similarly, the AV hierarchies themselves can either be hand-coded by the designer, or can be learned automatically. In Section 2.1.4, we give details on how we learn them automatically.

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 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. 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 ith 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 5 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 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 3, we 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. 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 QList. 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: $P(C) = \sum_{Q \in C} P(Q)$.

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 is 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. These coverage and overlap 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 single-

ton) 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 $2001 \rightarrow S_1 \wedge S_2$.

2.1.4 *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 ith 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 $d(Q_1, Q_2)$.[3] 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 2.1.3. The coverage and overlap statistics $P(\hat{S}|Q)$ for a specific query Q are computed using the statistics from the query list maintained by the mediator.

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

[3]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.

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 6), an agglomerative hierarchical clustering algorithm ([7]), 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*;

Fig. 6. 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 6, 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:

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

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

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.

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 the statistics of the specific queries[4] within these classes will be used to estimate the coverage of the sources for these general queries 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.

Flattening Attribute Value 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 levels of abstractions may not actually increase when we go up the hierarchy. For example, in Figure 7, assuming the three attribute values have the same (or very similar) 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 (or very similar) statistics multiple times.

In order to prune these unnecessary clusters, we use another algorithm called FAVH (Flattening AV Hierarchy, see Figure 8). FAVH starts the flattening procedure from the root of the AV Hierarchy, then recursively checks and flattens the whole 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) 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.

[4] 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.

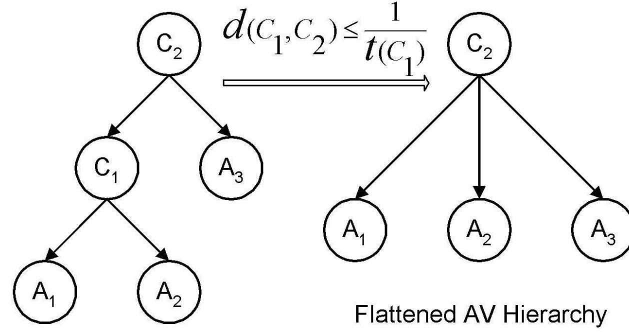


Fig. 7. An example of Flattening AV Hierarchy

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*;

Fig. 8. The FAVH algorithm

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.

2.1.5 *Discovering Frequent Query Classes.* Once the AV hierarchies are in place, the query class hierarchy is defined in terms of the cartesian product of the AV hierarchies. The statistics are stored with respect to these 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 more than the minimum frequency threshold *minfreq*. The example classes shown in Figure 5 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

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

Fig. 9. The DFC algorithm

a set of least-general ancestor classes which are not subsumed by each other (see Section 3.1). 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. Later in this section, we will then discuss how to prune candidate frequent classes with low class access probability.

Discovering Candidate Frequent Classes: We present an algorithm, DFC (Discovering Candidate Frequent Classes), (see Figure 9), to efficiently discover all the candidate frequent classes. The DFC algorithm dynamically prunes classes during counting and uses the *anti-monotone property*[5] [Han and Kamber 2000] 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,

[5] 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 ∈ Q)
        ftSet_i = {f_i};
        ftSet_i = ftSet_i ∪ ({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 ∉ classSet_i from tempSet;
            tempSet = tempSet × ftSet_{j_{i+1}};
        end for
        remove any class C ∉ classSet_{k−1} from tempSet;
        candidateSet = candidateSet ∪ tempSet;
    end for
    return candidateSet;
End genClassSet;
```

Fig. 10. Ancestor class set generation procedure

and so on. The algorithm continues until it gets all the candidate frequent classes with all the n features (where n is the total 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 10), where A_{c_i} is the feature value of the ith 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 5) . 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,

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

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 11), 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 3.1). 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 ∈ 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 ∈ mSet)
            FR_C ← FR_C + FR_Q;
            FR = FR + FR_Q;
        end for
    end for
    for(each class C)
        class access probability P_map(C) ← FR_C / FR;
    while ((∃C ∈ 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;
```

Fig. 11. The PLC procedure

2.1.6 *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 3.1, and update the corresponding coverage and overlap 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 [Agrawal and Srikant 1994] to avoid considering any supersets of an uncorrelated source set. We first identify individual sources with coverage statistics more than *minoverlap*, and keep coverage statistics for these sources. Then we discover all 2-*sourceSet* [6] with overlap more 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.

2.2 Mining Source Latency Statistics

In data integration scenarios, the speed of retrieving the answer tuples is also a very important measure of the goodness of a query plan. To optimize the execution cost, the mediator needs the source latency [7] statistics, since the sources differ significantly in their latency. Figure 12 shows the average time for retrieving the answer tuples from the 5 of the data sources using 200 queries randomly chosen from real user queries submitted to *Bibfinder*. It shows significant differences in query processing latency between the sources , which motivates our study on learning latency statistics of the sources and using them together with coverage/overlap statistics in our adaptive data integration framework.

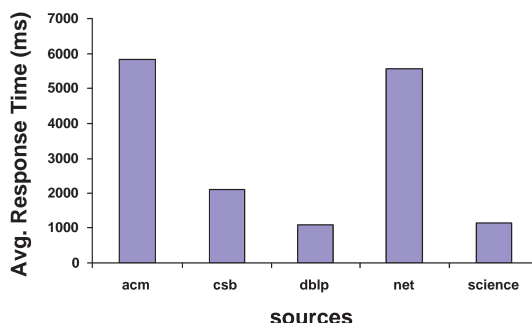


Fig. 12. Average response time for the 5 data sources in *Bibfinder*.

2.2.1 *Defining Latency (Response Time).* Because of the autonomous and dynamic nature of the data sources and the network topology, the latency statistics of individual sources have to be learned by the mediator. The first issue is how to define the latency. Some work has been done in learning source latency profiles. For example, [Gruser et al. 2000] defines latency (response time) as a source

[6] *k-sourceSet* denotes the source sets with only k sources.
[7] We use *latency* and *response time* interchangeably through out this paper.

specific measure with some variations on several dimensions (See detailed discussion in section 5). Such a definition assumes that the latency of a source is query insensitive and different queries submitted to a source have similar latency values (under similar settings, such as the time of a day they are submitted, and quantity of their answers). However such an assumption is questionable in many cases. For example when we monitored queries on *Bibfinder*, we found that the same data sources sometimes give very different latency values on different queries.

This observation suggests that to accurately estimate latency for the new queries, *the mediator has to learn the latency value in a query sensitive manner*. In other words, for any single data source, we have to learn latency values for different types of queries. This leads to the next challenge: how to properly classify the queries so that queries within the same category have similar latency values for that data source.

We already have the "query class" model from our coverage/overlap statistics, so a natural first idea would be to use such query classes to classify the queries and learn source specific latency values for each query class. However, the previous query class model is based on AV hierarchies and thus such a classification depends on the binding values of the queries. Based on our observations, the latency values of queries usually are not value sensitive, but rather more dependent on the binding pattern of the queries. Specifically, we found that for a given data source, the latency values are usually quite different when the binding pattern of the queries are different, as shown in Figure 13. At the same time, for queries with the same binding patterns, the latency of a given source is relatively stable, as shown in Figure 14. These observations suggest that the "query class" model defined in section 2.1.3 is not a good way to classify the queries in learning latency statistics. In stead, we decided to learn source latency statistics based on another type of "query classes" - the binding patterns of queries. For every [source, binding pattern] combination we learn the latency statistics and use them to predict the latency values of the future queries that are bound in the same pattern.

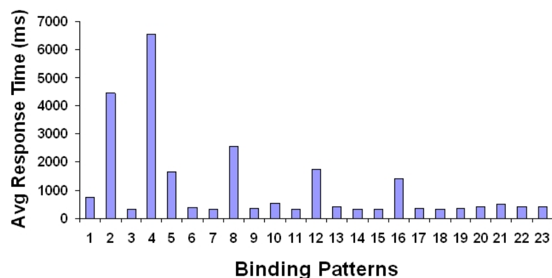


Fig. 13. The average latency of the source DBLP shows significant variance between different binding patterns.

Another minor issue is how to quantify the latency value. In the *Bibfinder* scenario, the latency is intended to measure how fast a source can answer a query. Some sources may export answers in multiple pages, but the time to retrieve the first page (which includes query processing time of the sources and the time for the

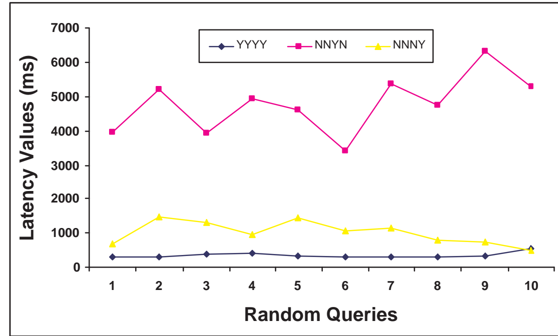


Fig. 14. The Latency values of the source DBLP with 10 randomly selected queries for each of the 3 different binding patterns. The queries that fall in same binding patterns have similar latency values.

mediator to parse the returned answer pages) is very short compared to the overall time. Obviously for users who care less about the completeness of the answer, the speed of retrieving the first page is more important. Moreover, when comparing different queries with the same binding patterns, we found the time to retrieve the first pages is reasonably stable. Thus, we decided to take this time as the latency value since it is more likely to be a good estimate for future queries. Admittedly, different data sources choose to export different number of answer tuples within each page, thus the time for the mediator to parse those pages would be different. However such differences in the size of pages can be ignored, as we noted that the source-side query processing time almost always significantly dominates the mediator-side parsing time. In another words, we consider the answers within a same page to have same latency values.

2.2.2 *Learning Latency Statistics.* As stated above, we need to learn the latency statistics for each [source, binding pattern] combination. Binding pattern refers to whether and how the attributes of a query are bound to concrete values. In the *Bibfinder* test bed, we found that for the attributes *title*, *conference* and *author*, whether or not these attributes are bound highly differentiates the latency for most data sources. For the attribute *year*, it is not a boolean choice, since it is a numeric attribute. We found that for a query with the *year* attribute bound to a range of numbers (e.g. 1994 to 2003), the latency is usually very different from that of a query with the *year* attribute bound to a single number, and both of them differ highly from the latency of a query that has the *year* attribute free. Therefore, in the *Bibfinder* scenario, there are $(2 \times 2 \times 2 \times 3 - 1)$, or 23 different patterns.

We randomly selected 20 real user queries for each of the binding patterns from the query log of *Bibfinder* and sent each of them to each of the 5 sources. We used the average response time as the latency value for each [source, binding pattern] combination. We conducted the probing several times at different times of the day to minimize the influence of the difference of source work load at different time of a day[8].

[8]In fact in contrast to the results in [Gruser et al. 2000], in our setting no obvious differences in

3. USING STATISTICS IN SOURCE SELECTION

With the learned statistics, the mediator is able to make source selection according to different user objectives. We will first discuss the use of statistics in single objective query optimization and then multi-objective optimization by combining multiple types of statistics.

Using latency statistics to select the fastest sources is a straightforward task, which is just a matter of ordering the learned latency statistics according to the binding patterns of the queries. In contrast, selecting the most relevant sources for a new query is not just a simple sorting of coverage statistics regarding that query, considering the inter-source overlap. In this section we will first discuss how to map a new query to a set of frequent query classes and use the coverage/overlap statistics of them to select the most relevant sources. Empirical evaluation will be presented to show the effectiveness of this approach.

3.1 Using Coverage/Overlap Statistics

With the learned coverage/overlap 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.

3.1.1 *Query Mapping.* Given a new query Q, we first get all the abstract values (features) from the AV hierarchies corresponding to the binding values (features) 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:

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

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 coverage and overlap statistics vectors of the mapped classes $\overrightarrow{P(\widehat{S}|C_i)}$ and their corresponding tightness

latency values at different times of the day are observed. The nature of the source servers and network topology largely decide the average response time of the sources.

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.

3.1.2 *Using Coverage and Overlap Statistics to Generate Query Plans.* Once we have the coverage and overlap statistics, we use the **Simple Greedy** and **Greedy Select** algorithms described in [Florescu et al. 1997] 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. A residual coverage computing algorithm is discussed in [Nie et al. 2003] to efficiently compute the residual coverage using the estimated coverage and overlap statistics. Specifically, recall that we only keep overlap statistics for correlated 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). 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. In particular, this algorithm, which exploits this structure of the stored statistics, 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.

3.1.3 *Experimental Setting.* We now describe the data, algorithms and metrics of our experimental evaluation on source selection using coverage/overlap statistics.

Database Set: 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 recent 25000 real queries asked by *Bibfinder* users as the query list as of May 20, 2003. Among them, we randomly chose 4500 queries as test queries and the others were used as training data. The AV Hierarchies for all of the four attributes were learned automatically using our GAVH algorithm. The learned Author hierarchy has more than 8000 distinct values[9], the Title hierarchy keeps only 1200 frequently asked keyword item sets, the Conference hierarchy has more than 600 distinct values, and the Year hierarchy has 95 distinct values. Note that we consider a range query (for example: ">1990") as a single distinct value.

Algorithms: In order to evaluate the effectiveness of our learned statistics, we implemented the **Simple Greedy** and **Greedy Select** algorithms described in [Florescu et al. 1997] to generate query plans using the learned source coverage and

[9]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.

overlap statistics. A simple **Random Select** algorithm is also used to randomly choose k sources as the top k sources.

Evaluation Metrics: We generate plans using the learned statistics and the algorithms mentioned above. The effectiveness of the statistics is estimated according to how good the plans are. The goodness of a plan, in turn, is evaluated by calling the sources in the plan as well as all the other sources available to the mediator. 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.

We also measure the *Estimation error* between the estimated statistics and the real coverage and overlap values. The *Estimation 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 ith 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.

3.1.4 *Experimental Results.* **Space Consumption for Different *minfreq* and *minoverlap* Thresholds**: In Figures 15 and 16, we observe the reduction in space consumption (and number of classes) when we increase the *minfreq* and *minoverlap* thresholds. As we can see in Figure 15, 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 16, 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[10].

Accuracy of the Learned Statistics for Different *minfreq* and *minoverlap* Thresholds: Figure 17 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 16 and 17 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

[10]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.

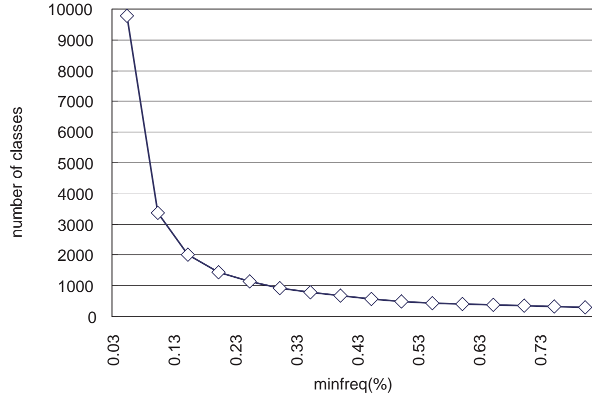


Fig. 15. The total number of classes learned

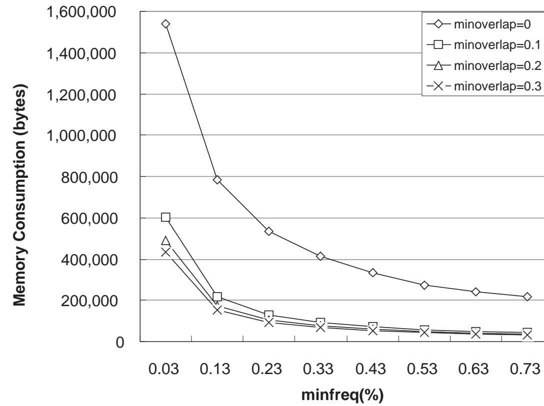


Fig. 16. The total amount of memory needed for keeping the learned statistics in BibFinder

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 algorithm, 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 18, 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,

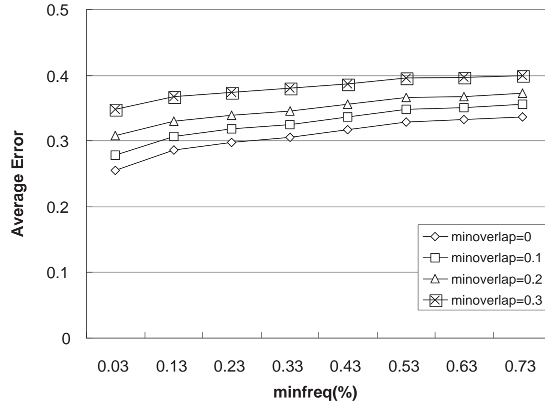


Fig. 17. The average distance between the estimated statistics and the real coverage and overlap values.

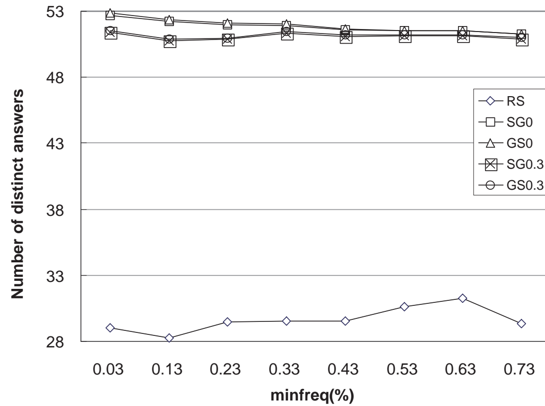


Fig. 18. The average number of answers *Bibfinder* returns by executing the query plans with top 2 sources.

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 19 and 20, 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 performance 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

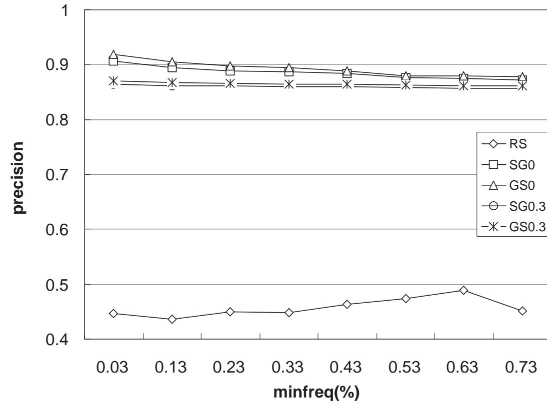


Fig. 19. Precision for query plans with top 2 sources.

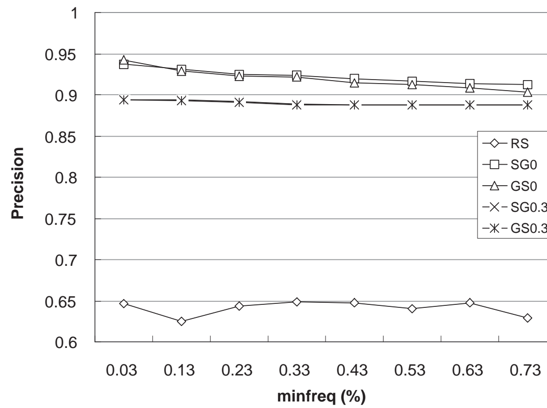


Fig. 20. Precision for query plans with top 3 sources.

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 previous experiment with a controlled data set, where we set 20 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. For detailed information about our experiments on the controlled data set, please see [Nie et al. 2003].

Figure 21 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

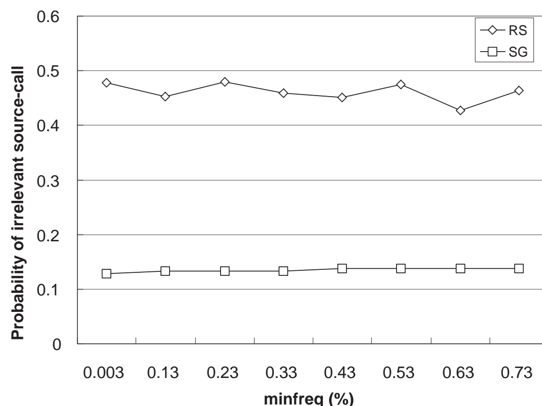


Fig. 21. The percent of the total source-calls that are irrelevant for query plans with top 1 sources.

reduce the unnecessary load on its integrated sources.

3.2 Multi-Objective Query Optimization

Using coverage/overlap statistics helps the query planner to generate a query plan that maximizes the coverage (number of answer tuples returned) with limited resources. However in most web-based data integration systems, coverage is usually not the only requirement of every user. For example in the *Bibfinder* scenario, when a user searches for bibliography information of a certain topic, she does not always care if the answer set is "complete", i.e. contains all the relevant publications. More often than not, in such systems users would like to get as many answers as fast as possible. In other words, they prefer a query plan that is optimal with respect to multiple objectives (in the example above, both *coverage* and *latency*).

Such multi-objective optimization entails two connected tasks: 1. collecting source statistics pertaining to the different objectives, and 2. combining them appropriately during query processing. Because of the nature of the autonomous data sources, such objectives are often conflicting. For example, some data sources may contain a large amount of data and are able to give high coverage for the given query, but they might have a longer latency. Some other sources may have less relevant tuples but they can respond very quickly. As we discussed earlier, for the efficiency and politeness reasons, the mediator always tries to direct a given query only to a subset of the data sources instead of all of them. Under such situations, the mediator has to be able to resolve such conflicts between multiple user objectives. Moreover, different users usually have different preferences on the different objectives. For example, some may rather get a few answers as fast as possible, whereas others may be willing to wait a little longer for higher coverage. Thus the joint optimization approach has to be flexible to adapt to different user preferences.

We investigated such multi-objective query optimization involving coverage and latency objectives in the *Bibfinder* scenario. We introduce a novel joint optimization model that can combine both coverage/overalap statistics and latency statistics to adapt to such multi-objective user requirements.

3.2.1 *Combining Multiple Objectives.* Now that we have both coverage/overlap statistics and latency statistics learned from past queries, we are able to jointly use them in source selection. When a new query is submitted to the mediator, the corresponding latency statistics of each source with respect to its binding pattern are retrieved. At the same time, the query is mapped to one or more frequent query classes to get the coverage/overlap estimation of it. In our current work, we experimented with the following source utility model which takes into account both the coverage/overalap statistics and the latency statistics:

$$Util(S_i) = ResidualCoverage(S_i|Q) \times \gamma^{Latency(S_i|Q)}$$

In the formula above, for a given query Q, the coverage estimate of source S_i on Q is discounted by the latency value of source S_i for queries that have same binding pattern as Q. The planner uses the utility value in a greedy way to select the sources to call. The source with the largest utility value is the first source added to the plan, and then the source with the next largest utility, and so on. Note that to maximize the overall coverage of the plan, we use *residual coverage* with respect to the set of already selected sources as discussed in Section 3.1. For the selection of the first source, the *residual coverage* is just the regular coverage value.

The utility function has the *discount factor* γ. We can see that when γ is 1, the utility value is just the coverage value, and the plan generated only optimizes on coverage. By reducing γ the sources that have shorter latency are favored and thus the plan is able to jointly optimize both coverage and latency. Adjusting γ makes it possible to flexibly adapt to different users' individual preferences on coverage and latency. We will show in next section that this combined utility function does make reasonable trade-offs between multiple objectives.

3.2.2 *Experimental Evaluation.* To evaluate the multi-objective query optimization approach, we experimented with *Bibfinder* to see if it can make reasonable trade-offs between latency and coverage and generate query plans that can reflect the users' preference for different objectives. We evaluated the utility function to see if the plans generated using combined statistics can reward both high coverage and low latency appropriately. For this purpose, we randomly chose 200 real user queries, made query plans using the discount formula, executed the plan and recorded the time to retrieve the first K tuples (K=1, 2, 3, ...). We varied the discount factor γ to see its influence in query planning. Figure 22 shows the average time for retrieving the first K (K = 1, 2, 3, ...) tuples with each of the curves representing the measure with different values of the discount factor γ. When γ is 1, we have a plan that only uses coverage/overlap statistics without considering the latency, and thus on average the speed of retrieving the first K answers is the lowest (because the faster sources are not rewarded for being faster). On the other hand, when the discount factor decreases, the coverage estimates of the sources are discounted with the latency, and the plans generated tend to favor the faster sources. As a result such plans are able to retrieve the first tuples faster on average.

This experimental evaluation shows that the latency values of different [source, binding pattern] combinations are a sound estimate of the future queries, and the joint optimization approach presented here is able to make flexible trade-offs according to the users' preferences regarding different objectives.

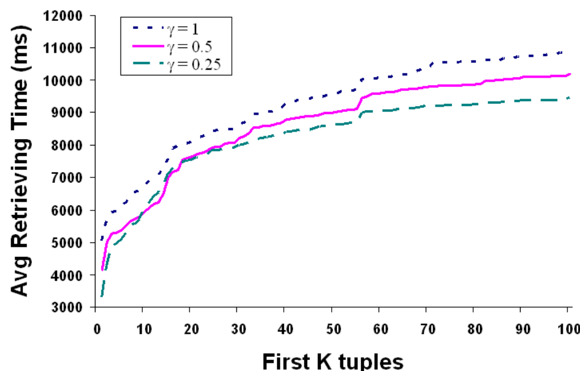


Fig. 22. Average time for retrieving first K tuples.

4. MAINTAINING SOURCE STATISTICS

Data integration systems usually evolve continuously, and the statistics of the sources thus have to be updated accordingly to keep their accuracy. For example, the user interest patterns might change over time and thus our frequency based coverage/overlap statistics have to be updated to capture the *recent* frequent query classes. Similarly, the data sources might upgrade their services and the network topology might change over time, and as a result the source latency statistics will also change accordingly.

All these changes bring up the need for maintaining statistics. In our framework, maintaining source latency statistics is straightforward. The mediator can always record the latency values of individual queries during its regular query processing. The mediator can periodically recompute the latency statistics using those of the recent queries in the query log. Such statistics recording and recomputing are straightforward and the cost of them is almost negligible. In contrast, updating the coverage/overlap statistics is more complicated and must be handled carefully so that the maintenance cost will not limit the scalability of the system. Hence in this section we will discuss the challenges of coverage/overlap statistics maintenance. We will then introduce a incremental maintenance approach to reduce the cost of maintenance but at the same time keep the accuracy of the statistics.

4.1 Motivation

The frequency based statistics mining approach of *Statminer* is a user adaptive one. It is efficient because the mediator keeps coverage/overlap statistics only for the "frequent query classes". In other words, more accurate information is kept for the query classes that users are more interested in, assuming that the access frequency represents the degree of user interest. When the user interest patterns change, the actual *frequent query class set* will change accordingly, i.e. some query classes might no longer be accessed frequently and some previously non-frequent query classes might suddenly get more attention. This is very common in the *Bibfinder* scenario because new "hot spots" in computer science emerge constantly.

When such shifts of user interest happen, using the old statistics without updating

would decrease the quality of statistics estimation for new queries. Thus over time statistics will be less effective in source selection. We will show empirical evidence for this kind of deterioration in Section 4.3. To keep statistics accurate over time, the mediator must change the previously mined *frequent query class set* to track the user interest.

A straightforward approach would be to periodically redo the learning process based on the entire query history of the mediator. There are at least two obvious drawbacks of this approach. First, although the cost of the learning process is linear in the number of queries in the query log, the log itself is large and keeps increasing. Second, learning from the entire history cannot effectively capture the shifts of recent user interest. The effectiveness of frequency based statistics mining approach is based on the assumption that we can predict the near future with the statistics learned from the immediate past. Thus, learning from the entire history will not be able to emphasize the importance of the immediate past as it treats the old (and thus less accurate) and new statistics equally. Specifically in our model, if statistics are learned from the entire history, then very few of the lower-level query classes (which represent finer granularity of approximation of queries) will be considered "frequent" in such a long history. The *frequent query class set* may contain mostly high level query classes, which are more coarse granularity abstraction of the individual queries. They are thus not able to approximate the new queries accurately, if the user interest is relatively concentrated on certain branches of the class hierarchy within a given period.

To overcome the problems above, a better way of updating the statistics is to learn the knowledge from a recent "time window" in the query log. The queries in this window are supposed to represent the recent past and the statistics learned from them are used to predict the source statistics of the near future. This will keep the learning costs under control and also capture the recent interests of the user population. An important issue is that of the window size. The size of the window should not be too small, because the effectiveness of any machine learning approach depends on a reasonably large training set to avoid learning too much noise. At the same time, the learning has to be done frequently enough to capture the shift of user interest in a timely manner. To meet both of these requirements, we introduce a "sliding window" update model as shown in Figure 23. In this model, the *window size* represents the recent fragment of the query log from which the statistics are mined. The *shift size* represents the time interval between consecutive updates. The *window size* needs to be large enough so that the learning technique can mine useful statistics and the *shift size* has to be small enough so that the recent change of user interests can be captured in a timely manner.

One remaining problem is that with a relatively large window size, the regular learning algorithm is still costly. Moreover, we can observe that when the window size is much larger than the shift size, there is a very big overlap between two consecutive windows, so the statistics we learn from two consecutive window may be very similar. Redoing the entire learning process on the new window would largely repeat the work that has been done in the learning cycle on the previous window. This observation motivates our investigation of the incremental learning algorithm. Specifically, we keep the statistics learned previously and use only the

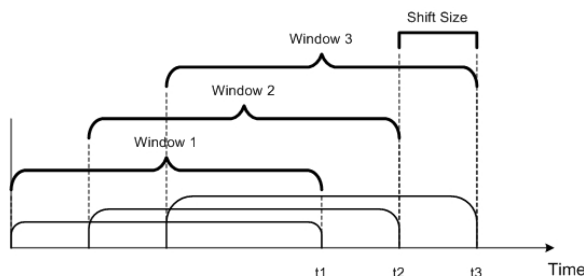


Fig. 23. Sliding window in query log.

queries in the "shift period" as the training data to calibrate the existing statistics and get the new statistics of the current window. Intuitively, since the $shift\ size$ is much smaller than $window\ size$, such training will be much less costly than regular learning on the entire window. More importantly, since it only partially changes the existing statistics, it is less likely to have the problem of introducing too much noise to the mined statistics.

In Section 4.3, we will show that incremental maintenance of the statistics will be less costly but still retains the quality of the statistics in terms of their effectiveness in query planning.

4.2 Approach of Incremental Statistics Maintenance

Our frequency based statistics mining approach controls the cost of learning and storage by using query classes as the abstraction of individual queries. Recall that the set of all query classes is the cartesian product of all the AV hierarchies of the classification attributes, and that the *frequent query class set* is just a subgraph of this cartesian product. As we discussed, the *frequent query class set* represents the queries that users are most interested in and thus we keep statistics on them. Updating the statistics requires changing the *frequent query class set* as the user interest patterns change.

The regular learning approach in our previous work maps every query in the training set to a group of query classes in the query class hierarchy and accumulates mapping frequencies of those classes. Only the classes that have a mapping frequency higher than a given threshold are kept as frequent classes. The statistics of the individual queries that are mapped to the frequent query classes are used to compute their statistics. This approach identifies the frequent query classes in the entire class hierarchy (cartesian product of all the AV hierarchies).

The reason for such learning being costly is, without prior knowledge of the distribution of "frequent query classes" in the entire class hierarchy, it needs to search globally in the space of query classes (cartesian product of all AV hierarchies) to identify the frequently accessed ones. The size of that space is exponential in the number of classification attributes and linear in the number of possible values of each classification, which are large numbers. For example, in the *Bibfinder* scenario, when we construct AV hierarchies based on the first $25,000$ queries, the size of the space of query classes is larger than 100 million, while the number of frequent query classes we identified (with certain thresholds) is around 1000. When searching for

frequent classes, although the regular learning has already cut the search space by taking advantage of the anti-monotone property [Han and Kamber 2000] of the class hierarchy, such searching still needs to look at many query classes.

When it comes to updating statistics, we already have the knowledge of the previous *frequent query class set*. To reduce the learning cost, the incremental learning algorithm has to exploit the previously learned statistics as much as possible and avoid such global search for frequent classes in the entire query class space.

The incremental learning approach tries to limit the space of searching for new frequent classes by starting the search from the existing frequent class hierarchy and modifying it. Specifically, if a query class C has been very frequently accessed lately but it is not currently considered a frequent class, then the closest ancestors of this class which are already in the frequent class set must be very frequently accessed too because the queries that belong to C are mapped to the ancestors instead. Similarly, if an existing frequent class is no longer frequently accessed, then its descendants that are also in frequent class set are not frequently accessed lately either. So, when searching for new frequent query classes, we can consider the children of the existing frequent class that are very frequently accessed as the candidates. For example, Figure 24 shows a very simple query class hierarchy and its change over time. In this hierarchy suppose we note that class C_1 is being accessed more frequently than expected. This might then lead us to further refine the hierarchy below C_1, adding C_2 as a new frequent class.

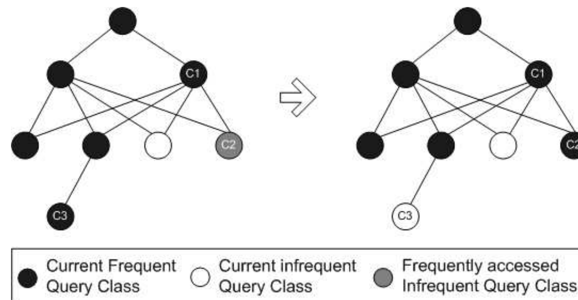


Fig. 24. A simple example showing the change of *frequent query class set*.

To implement such incremental modification of frequent query classes, the mediator needs to maintain the following information for each of the previously discovered frequent query classes:

—The source coverage and overlap statistics.

—The estimated access frequency of this class, which is the access frequency during the previous learning cycle and is used as the expected access frequency from then on.

—The real access frequency of this class since the previous learning cycle is over. This frequency is accumulated as the mediator keeps processing the new queries posted to it, and mapping the queries to the existing frequent classes to make query plans.

```
Procedure IUS(FQCS: Frequent Query Class Set;
QList: Query List of the recent window; T_split: Thresh-
old to Split; T_merge: Threshold to Merge; T_minFreq:
Minimum Access Frequency Threshold)
    addList = ∅;
    removeList = ∅;
    for (each class C ∈ FQCS)
        C.realFreq = C.numOfAccess / |QList|;
        if C.realFreq > C.estimatedFreq * T_split
            addList = addList ∪ C.children;
        else if C.realFreq < C.estimatedFreq / T_merge
            removeList = removeList ∪ {C};
    end for
    FQCS = (FQCS ∪ addList) − removeList;
    for (each query Q ∈ QList)
        Re-map Q to FQCS;
    end for
    for (each class C ∈ FQCS)
        if C.estimatedFreq < T_minFreq;
            FQCS = FQCS − {C};
    end for
End IUS;
```

Fig. 25. Incremental statistics maintenance procedure.

The mediator periodically invokes the incremental learning algorithm which compares the real access frequencies of the existing frequent classes and the estimated frequencies and uses the results of comparison as the basis to add or remove query classes into/from the *frequent query class set*. Figure 25 shows the algorithm IUS (Incrementally Updating Statistics) to perform such incremental statistics maintenance approach.

The algorithm first checks the real access frequency of each frequent query class and compares it with the estimated frequency. If a query class is very rarely accessed, i.e. the real frequency is much lower than the estimated one, then this class will be removed from the *frequent query class set*, as class C_3 shown in Figure 24. At the same time if the real frequency of a query class is much higher than the estimated one, that means the users are very interested in queries that fall into this query class. Thus the statistics about this query class should be further refined by introducing some of its subclasses into the *frequent query class set*.

In this case, we consider all the subclasses of this class as the potential candidates for new frequent classes and temporarily put all of them into the *frequent query class set*. After checking all of the classes, we re-map the queries in the query list onto the new *frequent query class set* and accumulate the new estimated frequency for each of them, and remove those newly added ones that are not really frequent. During this re-mapping, the statistics of the individual queries are combined to update the statistics of the already existing classes and compute the statistics of the newly added classes.

Once the learning process is over, the mediator resets the real access frequencies of the frequent query classes and starts to accumulate the values of them as it keeps

processing new queries submitted to it, until the next learning cycle is triggered. Therefore the mediator will be able to update the statistics continuously to keep up with changing user access patterns. Note that in Section 2.1.3 we defined *access frequency* as the number of queries mapped to it within a given period of time. In Algorithm IUS the values are normalized by the total number of queries in the recent window. The threshold values T_{split}, T_{merge} and $T_{minFreq}$ are set manually at present.

We can see that this incremental algorithm cuts the cost of searching for new frequent query classes by limiting the scope of modification to the vicinity of previous frequent query class hierarchy. Since we only inspect the *direct* children of current frequent classes instead of all their descendants, the new *frequent query class set* might not include all the classes that have been accessed with a frequency higher than given threshold. Therefore this approach may not be globally optimal in terms of identifying all the recent frequent classes. This of course results from our trade-off between optimality and cost of learning. However we will show in the next section that this approach cuts the cost significantly but does not sacrifice too much quality of the statistics.

4.3 Empirical Evaluation

We empirically evaluated the effectiveness of the incremental statistics maintenance approach on the *Bibfinder* test bed. It has been shown in Section 3.1.4 that the frequency based statistics mining approach is able to make quality source statistics estimations and source selection for new queries. We will first show that as time passes it is necessary to update the statistics to retain the accuracy of the statistics. We will then demonstrate that the incremental learning approach is able to reduce the learning cost and achieve quality statistics at the same time.

4.3.1 *Experiment Setting.* **Data Sources:** We use the same set of data sources in evaluating the regular statistics mining approach as in Section 3.1.4. We used the real user queries captured in the *Bibfinder* query log to train and test our algorithms. For the convenience of narration, we marked the queries in the order of the time when they were submitted to *Bibfinder*. Specifically the first query submitted to *Bibfinder* is marked as Q_1, and the next query is marked as Q_2, and so on. Naturally the time interval (or sets of consecutive queries) can be represented using the query number, for example $[Q_1, Q_{2000}]$ represents the period of time between Q_1 and Q_{2000}, as well as the set of queries that contains the queries from Q_1 to Q_{2000}.

Algorithms: In order to compare the effectiveness of incremental statistics maintenance algorithm and our previous non-incremental regular learning algorithm, we evaluated the different statistics learned from various [algorithm, training Set] combination. For example we use $S_{regular,[Q1,Q6000]}$ to denote the statistics mined by using the regular learning algorithm on a training set containing Q_1 to Q_{6000}.

Evaluation Metrics: We use the same evaluation metrics as shown in Section 3.1.4 to compare the statistics learned using regular learning approach and the incremental learning approach.

4.3.2 *Experimental Results.* **Measuring Change of User Interest:** Our frequency based statistics mining is user adaptive in the sense that we only learn and keep statistics for the frequent query classes, which represent the queries the user population is more interested in. Intuitively in data integration systems such as *Bibfinder*, user interest changes overtime. We tried to get a more quantified picture of whether and how these changes might happen. Using the same threshold values and regular learning algorithm, we identified 1129 frequent query classes from query list $[Q_1, Q_{25000}]$ and 458 frequent query classes from query list $[Q_{25000}, Q_{50000}]$. Among these 458 frequent query classes 84.5% of them are also present in the previous *frequent query class set*, but the other 15.5% are new frequent query classes. This shows a drift of user interest over time, which reinforces our motivation to update the learned statistics. On the other hand, the 84.5% remaining of query classes indicates the necessity of keeping and calibrating the old statistics instead of discarding them in the face of changes.

Necessity of Updating Statistics: To further illustrate the necessity of updating the statistics when user interest changes over time, we did the following experiment. We used the statistics $S_{regular,[Q_1,Q_{6000}]}$ as the baseline statistics. Then we updated the statistics using the "sliding window" model with a *window size* of 6000 and a *shift size* of 1000. For example, at the time of query Q_{7000}, we performed regular learning using $[Q_{1001}, Q_{7000}]$ as the training set, and incremental learning using $[Q_{6001}, Q_{7000}]$ as the training set. At the time of Q_{8000}, we performed regular learning using $[Q_{2001}, Q_{8000}]$ as the training set and incremental learning using $[Q_{7001}, Q_{8000}]$ as the training set, and so on. The difference is that regular learning involves rerunning the entire statistics learning process on the entire current window. In contrast, incremental learning takes advantage of the previously learned statistics and uses only the queries in the sliding period as the training set. For every 200 queries we calculated the average estimation error on the these queries, using the statistics $S_{regular,[Q_1,Q_{6000}]}$ (without update), current regularly updated statistics and current incrementally updated statistics respectively. Figure 26 shows the comparison of the average estimation errors until the time of Q_{13200}.

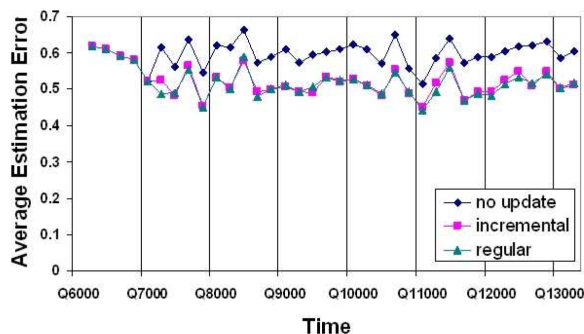


Fig. 26. Average estimation error using different statistics.

We can see that without updating the statistics, the average estimation error became higher as time passed. By updating the statistics the recent change in user interest is captured and the *frequent query class set* is updated accordingly. Consequently, the statistics estimation for the future queries is more accurate. Note that the *estimation error* is just the vector distance between the estimated statistics. Therefore, in Figure 26 the comparison only shows the differing accuracy of different sets of statistics, but the values themselves do not directly quantify the degree of accuracy for a single set of statistics (i.e. 0.7 does not mean the estimation is wrong in 70%).

Effectiveness of Incremental Updating: In our evaluation we compared the cost of learning statistics using both regular learning algorithm and incremental maintenance algorithm under the same experiment setting as shown above. Figure 27 shows the comparison of learning cost at each of the update points. It shows that with the given window size and shift size, the cost of incremental learning is always a very small fraction of that of regular learning (less than 20% overall). At the same time, we can see from Figure 26 that the incremental learning captures the change in the statistics very effectively. When used in estimating query statistics, the incrementally updated statistics reduced the estimation error almost as much as regular learning.

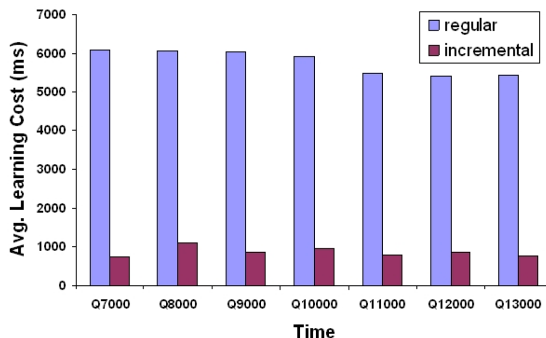


Fig. 27. Average learning cost.

One may argue that the in Figure 27, the learning cost of regular learning algorithm is not prohibitively high in terms of absolute values (around 6 seconds). However we must notice the fact that in our experimental setting, there are only 5 data sources. In large scale data integration systems, there would be much larger number of sources. As we stated above, the learning cost is exponential in the number of sources. Therefore in such systems, a decease of 80% in learning cost by using incremental maintenance algorithm will be very desirable.

Quality of Learned Statistics in Source Selection: Besides the estimation errors of the learned statistics themselves, another way to evaluate the quality of the statistics is to see the quality of the query plans generated by using them. We also experimentally evaluated the other two metrics, *plan coverage* and *plan precision*. We took the statistics learned from regular learning algorithm on $[Q_1, Q_{25000}]$ as the

baseline $S_{no-update}$, and used $[Q_{25001}, Q_{35000}]$ as the training set of regular learning to get updated statistics $S_{regular}$. We used the same training set for incremental update approach where *shift size* is 2000 (thus the statistics were updated 5 times incrementally at the time of Q_{35000}) to get the incrementally updated statistics $S_{incremental}$. We used queries of $[Q_{35001}, Q_{37000}]$ as the test queries and divided them into 5 test sets with 400 queries in each. We used the above three kinds of statistics to generate query plans (asking for top 2 sources) for the test queries and compared the average *plan coverage* and *plan precision* for each of the test sets.

Figure 28 shows the average *plan precision* of the 5 test sets. In all of the 5 test sets, using updated statistics results in query plans with higher average precision than using old statistics. Moreover, using incrementally maintained statistics in query planning arrives at plan precision that are almost as high as using statistics that are updated using regular learning approach, which costs significantly higher.

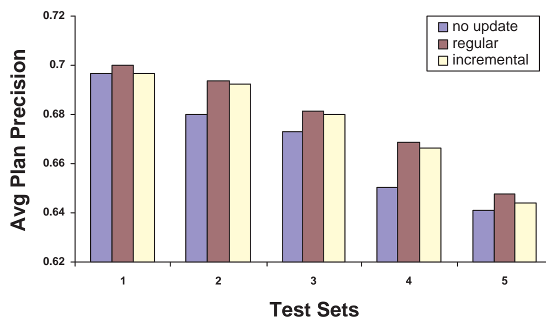


Fig. 28. Average plan precision using different statistics.

In summary, the comparison on the metrics above is consistent with our initial expectation of incremental statistics maintenance, i.e. the cost of learning is significantly lower but the quality of statistics learned is almost as good as learning everything from scratch periodically.

5. RELATED WORK

There has been little work on statistics gathering and source and user-profile learning in data integration scenarios. Although the utility of quantitative coverage statistics to rank the sources was explored in [Florescu et al. 1997] and [Doan and Halevy 2002], the primary aim of these effort was on the "use" of coverage statistics. [Naumann 2002] explored using qualitative source statistics in query processing information integration systems but did not discuss how such statistics are gathered. 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 [Zhu and Larson 1996] describe techniques for developing regression cost models for multi-database systems by selective querying. Adali et. al. [Adali et al. 1996] discuss how keeping track of rudimentary access statistics can help in doing cost-based optimizations.

In contrast to these efforts, in our research, we take a comprehensive look at learning, using and maintaining a wider range of statistics to support flexible query processing.

Learning Coverage/Overlap Statistics: some existing efforts consider the problem of text database selection [Ipeirotis and Gravano 2002; Wang et al. 2000] in the context of keyword queries submitted to meta-search engines. To calculate the relevance of a text database to a keyword query, most of the work ([Gravano and Garcia-Molina 1995; Xu and Callan 1998; Meng et al. 1999; Callan 2000]) uses the statistics about the document frequency of each single-word term in the query. The document frequency statistics are similar to our coverage statistics if we consider an answer tuple as a document. 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 users 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.

The similarity between document frequency statistics and coverage statistics suggests that an approach based on coverage and overlap statistics will also be beneficial in text databases. Indeed, recent work in our research group [Hernandez and Kambhampati 2005] adapted the ideas of *StatMiner* to the problem of text database ("collection") selection. The resulting approach has been shown to be superior to the traditional collection selection approaches such as CORI [Callan 2000; Callan et al. 1995].

The coverage/overlap statistics mining approach presented in this paper also differs from our previous work [Nie et al. 2004]. The approach in [Nie et al. 2004] depends on the domain experts to provide AV hierarchies. In contrast, in our current work we have developed methods to generate such AV hierarchies automatically based on the query log, which makes it more of a domain independent approach. Further, in [Nie et al. 2004], the "frequent query classes" are discovered based on the assumption that the frequency with which a query class is accessed is correlated with the size of that query class. In contrast, in our current work we present a improved approach that uses real query distribution to discover real *frequent* query classes (rather than *large* classes) and to learn statistics with respect to these classes. Finally, unlike [Nie et al. 2004], we also consider *learning multiple types of source statistics*, *multi-objective query processing* and *incremental statistics maintenance*.

Learning Latency Statistics: There has been some previous work on learning response-time statistics in data integration literature [Gruser et al. 2000; Viglas and Naughton 2002]. In contrast to the results of [Gruser et al. 2000], in our setting

the latency of a given query does not vary significantly in the dimensions of time of the day, day, and quantity of data, but rather is more dependent on its binding pattern. [Viglas and Naughton 2002] also pointed out the possibility of optimizing the time for retrieving first tuples instead of all tuples. Our approach differs from these efforts by using *query sensitive* source latency statistics together with other types of statistics in source selection to be adaptive to multiple user objectives.

Multi-Objective Optimization: Multi-objective optimization has been studied in database literature [Güntzer et al. 2000; Balke and Güntzer 2004] to use multiple data-oriented objective functions in ranking query results. In contrast, our work focuses on using source specific statistics in selecting relevant sources for given queries. In data integration scenarios, some existing query optimization approaches [Levy et al. 1996; Naumann et al. 1999; Doan and Levy 1999; Pottinger and Levy 2000; Balke and Güntzer 2004; Ives et al. 1999; Shanmugasundaram et al. 2000] use decoupled strategies by attempting to optimize multiple objectives in separate phases. Our own earlier work [Nie and Kambhampati 2001] studied joint optimization of multiple objectives during query planning with a weighed sum model, assuming the availability of source statistics in appropriate form. In this paper we develop the statistics learning model to actually gather such statistics, and a discounted model is used in combining coverage/overlap and latency statistics. Our empirical evaluation shows the general applicability of joint optimization of multiple objectives in data integration systems.

Incremental Statistics Maintenance: There has been some work on learning selectivity or sampling data in the relational database literature [Aboulnaga and Chaudhuri 1999; Ganti et al. 2000]. For example, [Aboulnaga and Chaudhuri 1999] introduced a self-tuning approach to progressively refine the histogram of selectivity of range selection operators. Our work is similar to it in the sense that both efforts try to infer the statistics from past query execution feedback instead of examining the data directly. However our work differs from it in at least the following aspects. First, our work aims to facilitate source selection in data integration systems, where query feedback is the only source of statistics learning. In contrast, [Aboulnaga and Chaudhuri 1999] *voluntarily* chose not to examine the data but to exploit query execution feedback for the concern of learning cost for a large scale relational database. Second, [Aboulnaga and Chaudhuri 1999] constructs the selectivity histogram for the range selection operators on single attributes, while our work introduces the query class hierarchy and directly learns statistics about the query classes which can have multiple attributes bound. Third, [Aboulnaga and Chaudhuri 1999] initializes the histogram with the assumption of uniform distribution of attribute values, but our work uses a past query list to learn the initial statistics. Last, our work tries to cut the statistics learning and storage cost by keeping statistics only with respect to the frequent query classes and updates the statistics according to recent user queries. As a result our work has an obvious user adaptive flavor. In this sense, our work is similar to [Ganti et al. 2000], which adjusts the probability of a tuple being used as a sample of the database also according to the recent knowledge of workload. It generates a group of samples to answer aggregation queries over a relational database. Our work differs from it by collecting different types of statistics (coverage and overlap) for different purposes

(source selection in data integration scenario).

6. CONCLUSION AND FUTURE WORK

In this paper we introduce an adaptive data integration framework that can effectively mine, use and maintain various types of source statistics. This framework is adaptive to the dynamic and autonomous nature of the data sources, various user preferences, as well as the shifting interest and access patterns of the user population.

We first presented the statistics mining approaches in this framework. We developed techniques that automatically learn source coverage/overlap statistics with respect to a set of frequently accessed query classes. We also developed a method to learn *query sensitive* source latency statistics. By using these statistics the query processor is able to make quality estimate for the new queries and optimize different query objectives (specifically, high coverage or low cost) respectively.

Users in data integration systems tend to prefer query plans that are optimal in terms of multiple objectives. To be adaptive to such multi-objective query processing requirements, we developed a novel multi-objective optimization model that can use both the coverage/overlap and latency statistics simultaneously to jointly optimize both the coverage and cost in source selection.

Moreover, to be adaptive to the dynamically changing system, we developed an incremental statistics maintenance algorithm to handle the change of interest in the user population. This approach significantly reduces the cost of statistics maintenance while retaining their accuracy.

All our techniques were evaluated in the context of *Bibfinder*, a fielded bibliography meta search engine. We provided compelling empirical evaluation result showing the effectiveness of these techniques in learning, using and maintaining a wider range of statistics to support flexible and adaptive query processing.

Our framework addresses two of the most important aspects of query processing in data integration systems: *coverage* and *latency*. Users of data integration systems may have requirements of query plans other than coverage and cost, such as data *density* (i.e., average fraction of null values in a tuple) [Naumann 2002] and *dominance* (an application-specific statistic such as "price" that gives preference information among tuples referring to the same entity), etc.. We will consider investigating the mining and using of those statistics in the future as well. As for the statistics maintenance, besides the change of user interests, there are many other types of changes in data integration systems that need to be handled. These include changes of source contents and addition/deletion of data sources into/from the systems. Our incremental coverage/overlap statistics maintenance solution is able to (partially and passively) capture the change of source content. This is because we use the queries in the recent time window to calibrate the old statistics. Consequently the change in source content that is projected on the recent queries is eventually integrated into the updated statistics. In the future we will consider handling such changes in a more complete and active way.

ACKNOWLEDGMENTS

We wish to thank Thomas Hernandez for his initial investigation and experiments on learning source latency statistics, and Ulnas Zambia for his many helpful suggestions and comments. This research is supported by ECR A601, the ASU Prop301 grant to ETI^3 initiative.

REFERENCES

ABOULNAGA, A. AND CHAUDHURI, S. 1999. Self-tuning histograms: Building histograms without looking at data. In *SIGMOD Conference*. 181–192.

ADALI, S., CANDAN, K. S., PAPAKONSTANTINOU, Y., AND SUBRAHMANIAN, V. S. 1996. Query caching and optimization in distributed mediator systems. In *SIGMOD Conference*. 137–148.

AGRAWAL, R. AND SRIKANT, R. 1994. Fast algorithms for mining association rules in large databases. In *VLDB*. 487–499.

BALKE, W.-T. AND GÜNTZER, U. 2004. Multi-objective query processing for database systems. In *VLDB*. 936–947.

BUNEMAN, P., KHANNA, S., AND TAN, W. C. 2001. Why and where: A characterization of data provenance. In *ICDT*. 316–330.

CALLAN, J. 2000. *Distributed Information Retrieval. In W.B. Croft, editor, Advances in information retrieval*. Kluwer Academic Publishers, Chapter 5, 127–150.

CALLAN, J. P., LU, Z., AND CROFT, W. B. 1995. Searching distributed collections with inference networks. In *SIGIR*. 21–28.

CITESEER. Computer and information science papers. http://www.citeseer.org.

DOAN, A. AND HALEVY, A. Y. 2002. Efficiently ordering query plans for data integration. In *ICDE*. 393–.

DOAN, A. AND LEVY, A. 1999. Efficiently ordering plans for data integration. *IJCAI-99 Workshop on Intelligent Information Integration*.

DUSCHKA, O. M., GENESERETH, M. R., AND LEVY, A. Y. 2000. Recursive query plans for data integration. *J. Log. Program. 43,* 1, 49–73.

FLORESCU, D., KOLLER, D., AND LEVY, A. Y. 1997. Using probabilistic information in data integration. In *VLDB*. 216–225.

GANTI, V., LEE, M.-L., AND RAMAKRISHNAN, R. 2000. Icicles: Self-tuning samples for approximate query answering. In *VLDB*. 176–187.

GRAVANO, L. AND GARCIA-MOLINA, H. 1995. Generalizing gloss to vector-space databases and broker hierarchies. In *VLDB*. 78–89.

GRUSER, J.-R., RASCHID, L., ZADOROZHNY, V., AND ZHAN, T. 2000. Learning response time for websources using query feedback and application in query optimization. *VLDB J. 9,* 1, 18–37.

GÜNTZER, U., BALKE, W.-T., AND KIESSLING, W. 2000. Optimizing multi-feature queries for image databases. In *VLDB*. 419–428.

HAN, J. AND KAMBER, M. 2000. *Data Mining: Concepts and Techniques*. Morgan Kaufmman Publishers.

HERNANDEZ, T. AND KAMBHAMPATI, S. 2005. Improving text collection selection with coverage/overlap statistics. In *The 14th International World Wide Web Conference*. Poster.

IPEIROTIS, P. G. AND GRAVANO, L. 2002. Distributed search over the hidden web: Hierarchical database sampling and selection. In *VLDB*. 394–405.

IVES, Z. G., FLORESCU, D., FRIEDMAN, M., LEVY, A. Y., AND WELD, D. S. 1999. An adaptive query execution system for data integration. In *SIGMOD Conference*. 299–310.

LAMBRECHT, E., KAMBHAMPATI, S., AND GNANAPRAKASAM, S. 1999. Optimizing recursive information-gathering plans. In *IJCAI*. 1204–1211.

LEVY, A. Y., RAJARAMAN, A., AND ORDILLE, J. J. 1996. Querying heterogeneous information sources using source descriptions. In *VLDB*. 251–262.

MENG, W., LIU, K.-L., YU, C. T., WU, W., AND RISHE, N. 1999. Estimating the usefulness of search engines. In *ICDE*. 146–153.

NAUMANN, F. 2002. *Quality-driven query answering for integrated information systems.* Springer-Verlag New York, Inc., New York, NY, USA.

NAUMANN, F., FREYTAG, J. C., AND LESER, U. 2004. Completeness of integrated information sources. *Inf. Syst. 29,* 7, 583–615.

NAUMANN, F., LESER, U., AND FREYTAG, J. C. 1999. Quality-driven integration of heterogenous information systems. In *VLDB*. 447–458.

NIE, Z. AND KAMBHAMPATI, S. 2001. Joint optimization of cost and coverage of query plans in data integration. In *CIKM*. 223–230.

NIE, Z. AND KAMBHAMPATI, S. 2004. A frequency-based approach for mining coverage statistics in data integration. In *ICDE*. 387–398.

NIE, Z., KAMBHAMPATI, S., AND HERNANDEZ, T. 2003. Bibfinder/statminer: Effectively mining and using coverage and overlap statistics in data integration. In *VLDB*. 1097–1100.

NIE, Z., KAMBHAMPATI, S., AND NAMBIAR, U. 2004. Effectively mining and using coverage and overlap statistics for data integration. *IEEE Transactions on Knowledge and Data Engineering*.

POTTINGER, R. AND LEVY, A. Y. 2000. A scalable algorithm for answering queries using views. In *VLDB*. 484–495.

SHANMUGASUNDARAM, J., TUFTE, K., DEWITT, D. J., MAIER, D., AND NAUGHTON, J. F. 2000. Architecting a network query engine for producing partial results. In *WebDB (Selected Papers)*. 58–77.

VIGLAS, S. AND NAUGHTON, J. F. 2002. Rate-based query optimization for streaming information sources. In *SIGMOD Conference*. 37–48.

WANG, W., MENG, W., AND YU, C. T. 2000. Concept hierarchical based text database categorization in a metasearch engine environment. In *WISE*. 283–290.

XU, J. AND CALLAN, J. P. 1998. Effective retrieval with distributed collections. In *SIGIR*. 112–120.

ZHU, Q. AND LARSON, P.-Å. 1996. Developing regression cost models for multidatabase systems. In *PDIS*. 220–231.