

A Frequency-based Approach for Mining Coverage Statistics in Data Integration

Zaiqing Nie & Subbarao Kambhampati
Department of Computer Science and Engineering
Arizona State University
Tempe, AZ 85287-5406
Email: {nie, rao}@asu.edu
ASU CSE TR-03-004

Abstract

Query optimization in data integration requires source coverage and overlap statistics. Gathering and storing the required statistics presents many challenges, not the least of which is controlling the amount of statistics learned. In this paper we introduce StatMiner, a novel statistics mining approach which automatically generates attribute value hierarchies, efficiently discovers frequently accessed query classes based on the learned attribute value hierarchies, and learns and stores statistics only with respect to these classes. We describe the details of our method, and present experimental results demonstrating the efficiency and effectiveness of our approach. Our experiments are done in the context of BibFinder, a publicly available bibliography mediator.

1 Introduction

The availability of structured information sources on the web has recently lead to significant interest in query processing frameworks that can integrate information sources available on the Internet. Data integration systems [LRO96, ACPS96, LKG99, DGL00, PL00, NKH03] are being developed to provide a uniform interface to a multitude of information sources, query the relevant sources automatically and restructure the information from different sources. In a data integration scenario, a user interacts with a mediator system via a mediated schema. A mediated schema is a set of virtual relations, which are effectively stored across multiple and potentially overlapping data sources, each of which only contain a partial extension of the relation. Query optimization in data integration [FKL97, NLF99, NK01, DH02] thus requires the ability to figure out what sources are most relevant to the given query, and in what order those sources should be accessed. For this purpose, the query optimizer needs to access statistics about the coverage of the individual sources with respect to the given query, as well as the degree to which the answers they export overlap. Gathering these statistics presents several challenges because of the autonomous nature of the data sources. In this paper, we motivate and investigate the issues involved in statistics gathering

in the context of a bibliography mediation system that we are developing called *BibFinder*.

BibFinder Scenario: We have been developing *BibFinder* (Figure 2, <http://rakaposhi.eas.asu.edu/bibfinder>), a publicly “fielded” computer science bibliography mediator. *BibFinder* integrates several online Computer Science bibliography sources. It currently covers *CSB*, *DBLP*, *Network Bibliography*, *ACM Digital Library*, *ScienceDirect*, and *CiteSeer*. Plans are underway to add several additional sources including *IEEE Xplore*, *ACM Guide*, *AMS MathSciNet* and *Computational Geometry Bibliography*. Since its unveiling in December 2002, *BibFinder* has been getting on the order of 200 queries a day. Most of the queries are “selection” queries on the papers.

The sources integrated by *BibFinder* are autonomous and partially overlapping. By combining the sources, *BibFinder* can present a unified and more complete view to the user. However it also brings some interesting optimization challenges. The global schema exported by *BibFinder* can be modeled in terms of just the relation: **paper(title, author, conference/journal, year)**. Each of the individual sources may export only a subset of the global relation. For example, the source *Network Bibliography* only contains publications in Networks, *DBLP* gives more emphasis to Database related publications, while *ScienceDirect* has only archival journal publications.

Need for 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 $Q(\text{title,author}) :- \text{paper}(\text{title, author, conference/journal, year}, \text{conference/journal} = \text{“SIGMOD”})$.

The 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* is most relevant, and *Network Bibliography* is much less relevant. Furthermore, since *DBLP* stores records of virtually

all the SIGMOD papers, a call to *Citeseer* is largely redundant.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 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 next best rank in terms of coverage but has minimal *overlap* (common tuples) with S' .

The Costs of Statistics Learning: 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.

Motivation for Frequency-based Statistics Learning: One way of keeping both these costs down is to learn statistics only with respect to a smaller set of “frequently asked” queries that are likely to be most useful in answering user queries. This strategy trades accuracy of statistics for reduced statistics learning/storing costs. In the *BibFinder* scenario, for example, we could learn statistics with respect to the list of queries that are actually posed to the mediator over a period of time. *BibFinder* facilitates this by maintaining a log of queries, and for each query keeping statistics on how many of the query answers came from which sources. The motivation of such an approach is that even if a mediator cannot provide accurate statistics for every possible query, it can still achieve a reasonable average accuracy by keeping more accurate coverage and overlap statistics for queries that are asked more frequently, and less accurate statistics for infrequent queries. The effectiveness of this approach is predicated on the belief that in most real-world scenarios, the distribution of queries posed to a mediator is not *uniform*, but rather Zipfian. This belief is amply validated in *BibFinder*. Figure 1 shows the distribution of the keywords, and bindings for the Year attribute used in the first 15000 queries that were posed to *BibFinder*. Figure 1(a) shows that the

¹In practice, *Citeseer* is not completely redundant since it often provides additional information about papers, such as pdf files 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 her query. 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.

most frequently asked 10% keywords appear in almost 60% of all the selection queries binding attribute Title. Figure 1(b) shows that the users are much more interested in recently published papers.

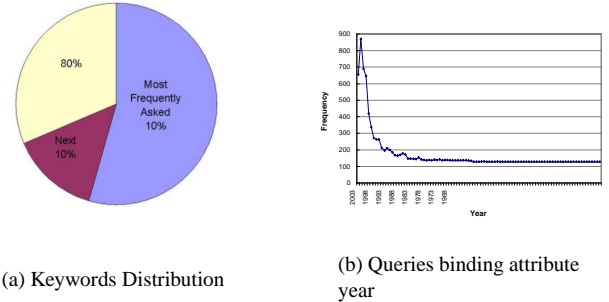


Figure 1. Query Distributions in BibFinder

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

Modeling Query Classes: The foregoing discussion about query classes raises the issue regarding the way query classes are defined 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 4 shows an example of AV hierarchies and the corresponding query classes (see Section 2 for details). AV hierarchies could be hand-developed or automatically generated (see Section 3) 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

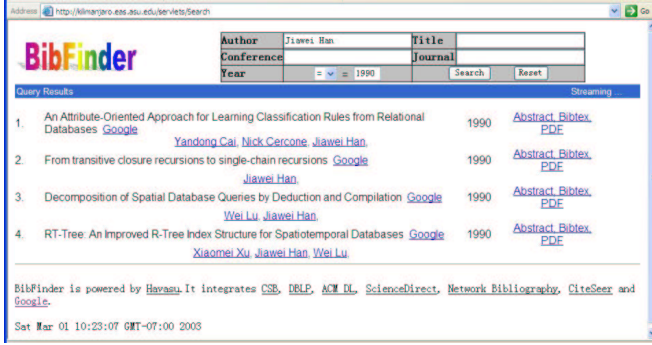


Figure 2. The BibFinder User Interface

or both of those attributes (to possibly general values of the attribute).²

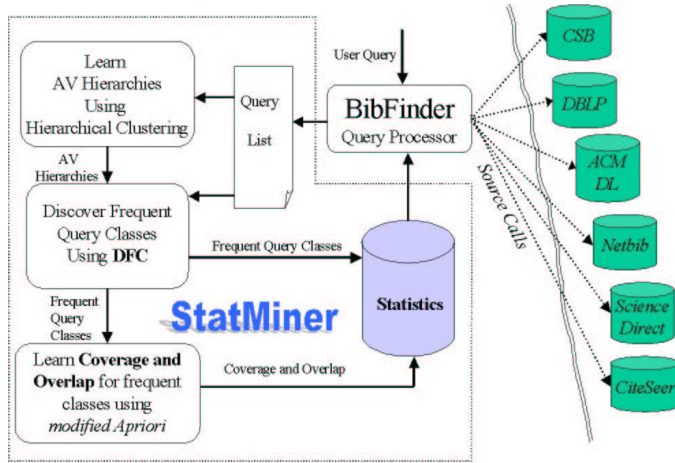


Figure 3. StatMiner Architecture

The approach to statistics learning described and motivated in the foregoing has been implemented in *StatMiner*, and has been evaluated in the context of *BibFinder*. In this paper, we describe the details of the *StatMiner* approach, and its use in *BibFinder*. Figure 3 shows the high-level architecture of *StatMiner*. *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

²This 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!

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 information source *DBLP* covers all the paper information for *SIGMOD* related queries. When the mediator encounters a new query Q , it is first mapped to a set \mathcal{C} of closest least-general query classes. The source coverage and overlap statistics for Q are computed as a (weighted) combination of the statistics of the query classes in \mathcal{C} . Notice that \mathcal{C} can be a singleton if the system happens to have statistics for Q directly. In this case we have fully accurate coverage and overlap statistics about Q . In all other cases, the statistics are approximate.

The rest of the paper is organized as follows. In the next section, we define some terminology about query classes and AV hierarchies. Section 3 describes the details of learning AV hierarchies. Section 4 describes how query classes are formed by taking the product of AV hierarchies and efficiently pruning the infrequent ones. Section 5 describes how coverage and overlap statistics are learned for the query classes that are retained. Section 6 describes how a new query is mapped to the appropriate query classes, and how the combined statistics are used to develop a query plan (i.e. the plan of accessing the sources). Section 7 describes the setting for the experiments we have done with *StatMiner* and *BibFinder* to evaluate the effectiveness of our approach. Section 8 presents the experimental results. Section 9 discusses related work and possible extensions, and Section 10 presents our conclusions.

2 AV Hierarchies and Query Classes

AV Hierarchy: As we are considering selection queries, we can classify the queries in terms of the selected attributes and their values. To abstract the classes further we assume that the mediator has access to the so-called “attribute value hierarchies” for a subset of the attributes of each mediated relation. An AV hierarchy (or attribute value hierarchy) over an attribute A is a hierarchical classification of the values of the attribute A . The leaf nodes of the hierarchy correspond to specific concrete values of A , while the non-leaf nodes are abstract values that correspond to the union of values below them. Figure 4 shows two very simple AV hierarchies for the “conference” and “year” attributes of the “paper” relation. Note that hierarchies do not have to exist for every attribute, but rather only for those attributes over which queries are classified. We call these 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 3, 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

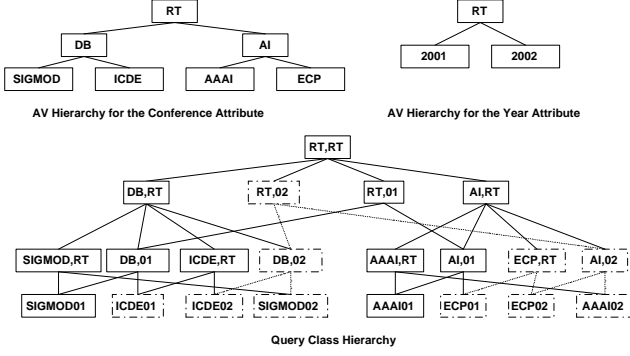


Figure 4. AV Hierarchies and the Corresponding Query Class Hierarchy

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 i^{th} classificatory attribute. Then the set of all query classes (called *classSet*) is simply $H_1 \times H_2 \times \dots \times H_n$. 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 belongs 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 4 shows an example class hierarchy for a very simple mediator with the 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 5, we show a query list fragment. The statistics we remember in the query list are: (1) query frequency, (2) total number of distinctive answers from all the sources (i.e. $|Answers|$) and (3) number of answers from each source set which has answers for the 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 . It can be computed using the formula: $P(Q) = \frac{FR_Q}{FR}$. The *class probability* of a 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: $P(C) = \sum_{Q \in C} P(Q)$.

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

Figure 5. A Query List Fragment

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 \hat{S} of sources with respect to a query Q , denoted by $P(\hat{S}|Q)$, is the probability that a random answer tuple of the query Q is present in each source $S \in \hat{S}$. The overlap (or coverage when \hat{S} is a singleton) statistics w.r.t. a query Q are computed using the following formula

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

Here $N_Q(\hat{S})$ is the number of answer tuples of Q that are in all sources of \hat{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.

The *coverage* of a source S w.r.t. a class C , denoted by $P(S|C)$, is the probability that a random answer tuple of a random query belonging to the class C is present in source S . The *overlap* among a set \hat{S} of sources with respect to a class C , denoted by $P(\hat{S}|C)$, is the probability that a random answer tuple of a random query belonging to the class C is present in each source $S \in \hat{S}$. The overlap (or coverage when \hat{S} is a singleton) statistics w.r.t. a query class C can be computed using the following formula:

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

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.

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 $C \rightarrow \hat{S}$, where C is a query class, and \hat{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 \hat{S} is a singleton) statistic $P(\hat{S}|C)$ is simply the *confidence* of such an association rule (denoted by $P(\hat{S}|C) = \frac{P(C \cap \hat{S})}{P(C)}$). Examples of such association rules include: $AAAI \rightarrow S_1$, $AI \rightarrow S_1$, $AI \& 2001 \rightarrow S_1$ and $2001 \rightarrow S_1 \wedge S_2$.

3 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 the complications of our 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 ($Q1, Q2$):

$$d(Q1, Q2) = \sqrt{\sum_i [P(\hat{S}_i|Q1) - P(\hat{S}_i|Q2)]^2}$$

Where \hat{S}_i denotes the i^{th} source set of all possible source sets in the mediator. Although the number of all possible source sets is exponential in terms of the number of available sources, we only need to consider source sets with answers for either of the two queries to compute $d(Q1, Q2)$ ³. The interpretation of the distance function is that we consider two queries similar if their source coverage and overlap statistics are similar. Similarly we define a distance function to measure the distance between a pair of query classes (C_1, C_2):

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

We compute a query class's coverage and overlap statistics $P(\hat{S}|C)$ according to the definition of the overlap (or coverage) w.r.t. to a class given in Section 2. 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.

³For example, suppose Query $Q1$ gets tuples from only sources $S1$ and $S5$, and $Q2$ gets tuples from $S5$ and $S7$, we will only consider source set $\{S1\}, \{S5\}, \{S1, S5\}, \{S7\}$, and $\{S5, S7\}$.

3.1 Generating AV Hierarchies

For now we will assume that all attributes have a discrete set of values, and we will also assume that the corresponding coverage and overlap statistics are available (see the last two paragraphs in this subsection for detailed discussion). We now introduce GAVH (Generating AV Hierarchy, see Figure 6), an agglomerative hierarchical clustering algorithm ([HK00]), to automatically generate an AV Hierarchy for an attribute.

Algorithm GAVH()

for (each attribute value)

generate a cluster node C ;

feature vector $C.fv = \overrightarrow{(P(\hat{S}|Q), P(Q))}$;

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 ;

generate a new cluster C ;

$C.fv = \overrightarrow{(\frac{P(C_1) \times P(\hat{S}|C_1) + P(C_2) \times P(\hat{S}|C_2)}{P(C_1) + P(C_2)}, P(C_1) + P(C_2))}$;

$C.children = (C_1, C_2)$;

put cluster C into AVQueue;

remove cluster C_1 and C_2 from AVQueue;

end while

End GAVH;

Figure 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: $\overrightarrow{(P(\hat{S}|C), P(C))}$, while $P(\hat{S}|C)$ 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 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 $\overrightarrow{P(\hat{S}|C)}$ by using the feature vectors of its children clusters C_1, C_2 :

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

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

Attribute Value Preprocessing: 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 attribute Title in *BibFinder* we learn the frequently asked keywords using an item set mining algorithm. Keyword sets that are rarely asked will not be remembered as attribute values. Each frequent keyword set will be treated as a discrete attribute value.

Handling Low Selectivity Attribute Values: If an attribute value (i.e. a selection query binding value) is too general, some sources may only return a subset of answers to the mediator, while others may not even answer such general queries. In such cases the mediator is unable to figure out the real number of tuples in these sources, and thus cannot know the coverage and overlap statistics of these queries to generate AV hierarchies. Hence we have to use the coverage statistics of more specific queries 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 their sub-queries will be used to estimate the coverage of the sources for these query classes using the following formula:

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

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 sub-queries for the class and ignore the influence of the general sub-queries.

3.2 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 of 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 completely (or almost) 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. The *tightness* $t(C)$, of a cluster C , is calculated as following:

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

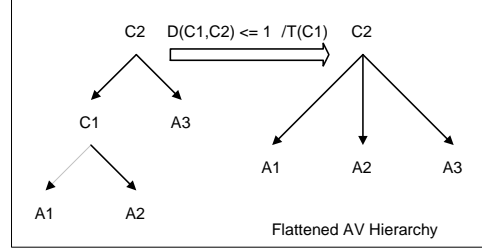


Figure 7. An example of Flattening AV Hierarchy

```

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 ( $d(C_{child}, C) \leq \frac{1}{t(C_{child})}$ )
      put ( $C_{child}$ ).children into Children_Queue;
      remove  $C_{child}$  from Children_Queue;
    end if
  for (each children node  $C_{child}$  in Children_Queue)
    FAVH( $C_{child}$ );
  end if
End FAVH;
  
```

Figure 8. The FAVH algorithm

while $d(Q, C)$ is the distance between the query Q and the cluster.

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

4 Discovering Frequent Query Classes

As we discussed earlier, it may be prohibitively expensive to learn and keep in memory the coverage and overlap statistics for every possible query class. In order to keep the number of association rules low, we would like to prune query classes which are rarely asked. We use a threshold on the support of a class (i.e. percentage of total frequency of queries that use the statistics of that class), called *minfreq*, to identify frequent query classes. Coverage and overlap statistics are learned only with respect to these frequent classes.

Candidate Frequent Classes and Class Access Probability: 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 4 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 not subsumed by each other. For example, the query (or class) (ICDE,01) has both the class (DB,01) and class (ICDE,RT) as its parent class. For a query with multiple ancestor classes, we need to map the query into a set of least-general ancestor classes which are not subsumed by each other (see Section 6). 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 can be computed using the following formula:

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

4.1 Discovering Candidate Frequent Classes

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 this subsection, we will introduce an algorithm to discover candidate frequent classes. In the next subsection, we will then discuss how to prune candidate frequent classes with low class access probability.

We now 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*⁴ ([HK00]) to avoid generating classes which are supersets of the pruned classes.

Specifically the algorithm makes multiple passes over the query list $QList$. It first finds all the candidate frequent classes with just one feature, then it finds all the candidate frequent classes with two features using the previous results and the anti-monotone property to efficiently prune classes before it starts counting, and so on. The algorithm continues until it gets all the candidate frequent classes with all the n features (where n is the total number of classificatory attributes). 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 algorithm finds all the candidate ancestor classes with k features for a query $Q = \{A_{c_1}, \dots, A_{c_n}, frequency\}$ by procedure **genClassSet**(see Figure 10), where A_{c_i} is the feature value of the i^{th} classificatory attribute. The procedure prunes infrequent classes using the frequent class set $classSet$ found in the previous $(k - 1)$ passes. In order to improve the efficiency of the algorithm, it dynamically prunes infrequent classes during the cartesian product procedure.

We explain the procedure using the following example.

⁴If a set cannot pass a test, all of its supersets will fail the same test as well.

Algorithm DFC($QList$; $minfreq$: minimum support; n : # of classificatory attributes)

```

classSet = {};
for(k = 1; k <= n; k + 1)
  Let classSetk = {};
  for(each query Q ∈ QList)
    CQ = genClassSet(k, Q, ...);
    for(each class c ∈ CQ)
      if(c ∉ classSetk)
        then classSetk = classSetk ∪ {c};
           c.frequency = c.frequency +
Q.frequency;
    end for
  end for
  classSetk = {c ∈ classSetk | c.frequency >=
minfreq};
  classSet = classSet ∪ classSetk;
end for
return classSet;
End DFC;
```

Figure 9. The DFC algorithm

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 4). 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 ,

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

In the DFC algorithm, we assume that the number of classes will be high. In order to avoid considering a large number of classes, we prune classes during counting. By doing so, we have to scan the dataset n times, where n is the number of classificatory attributes. The number of classes we can prune will depend on the threshold. A very low threshold will not benefit too much from the pruning. In the worst case where the threshold is equal to zero, we still have to preserve all the classes ($\prod_{i=1}^n |H_i|$, where H_i is the i^{th} AV hierarchy.). However if the number of classes is small and the cost of scanning the whole dataset is very expensive, then we can use a one pass algorithm. For each query Q , the algorithm has to generate a complete candidate class set of Q , and increase the frequency of each class in the set. By doing so, we have to remember the frequencies for all the possible classes during the

```

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

```

Figure 10. Ancestor class set generation procedure

```

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

```

Figure 11. The PLC procedure

counting.

4.2 Pruning Low Access Probability Classes

The DFC algorithm will discover all the candidate frequent classes, which unfortunately may include many infrequently mapped classes. Here we introduce another algorithm, PLC (Pruning Low Access Probability Classes), (see Figure 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 6). 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*.

5 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, and update the corresponding coverage and overlap statistics vectors $\overrightarrow{P(\hat{S}|C)}$ of its ancestor classes

using the query's coverage and overlap statistics vector $\overrightarrow{P(\hat{S}|Q)}$ through the formula $\overrightarrow{P(\hat{S}|C)} = \frac{\sum_{Q \in C} \overrightarrow{P(\hat{S}|Q)} \times P(Q)}{P(C)}$. 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 re-member the coverage and overlap statistics for each source set and class combination. If the mediator has access to *n* sources and has discovered *m* frequent classes, then the memory requirement for learning these statistics is $m \times 2^n \times k$, where *k* is the number of bytes needed to store a float number. If $k = 1$, $m = 10000$, and the total number of memory available is 1GB, this approach would not scale well when the number of sources is greater than 16.

In order to handle scenarios with large number of sources, we use a modified Apriori algorithm ([AS94]) to avoid considering any supersets of an uncorrelated source set.

We first identify individual sources with coverage statistics more than *minoverlap*, and keep coverage statistics for these sources. Then we discover all 2-sourceSet⁵ 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. This procedure can use the anti-monotone property to avoid considering any supersets of an uncorrelated source set.

⁵*k*-sourceSet denotes the source sets with only *k* sources.

6 Using Learned Coverage and Overlap Statistics

With the learned statistics, the mediator is able to find relevant sources for answering a coming 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.

6.1 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 $cSet$:

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

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

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

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

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

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

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

6.2 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 [FKL97] to generate query plans. Specifically, *Simple Greedy* generates plans by greedily selecting the top k sources ranked only

according to their coverages, while *Greedy Select* selects sources with high residual coverages calculated using both the coverage and overlap statistics.

Specifically we discuss how we compute the residual coverages in order to rank the sources for the new query Q using the estimated statistics. In order to find a plan with top k sources, we start by selecting the source with the highest coverage as the first source. We then we use the overlap statistics to compute the residual coverages of the rest of the sources to find the second best, given the first; the third best, given the first and second, and so on, until we get a plan with the desired coverage.

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

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

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

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

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

A naive evaluation of this formula would require 2^n accesses to the database of learned statistics, corresponding to the overlap of each possible subset of the n sources with source S . It is however possible to make this computation more efficient by exploiting the structure of the stored statistics. Specifically, recall that we only keep overlap statistics for 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). Furthermore, if the overlap is zero for a source set \widehat{S} , we can ignore looking up the overlap statistics for supersets of \widehat{S} , since they will all be zero by the anti-monotone property.

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

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

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

In Figure 12, we present an algorithm that uses this structure to evaluate the residual coverage in an efficient fashion. In particular, this algorithm will cut the number of statistics lookups from 2^n to $\mathcal{R} + n$, where \mathcal{R} is the total number of overlap statistics remembered for class C and n is the total number of sources already selected. This consequent efficiency is critical in practice

```

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

```

Figure 12. Algorithm for computing residual coverage

since computation of residual coverage forms the inner loop of any query processing algorithm that considers source coverage.

The inputs to the algorithm in Figure 12 are the source s for which we are going to compute the residual coverage, and the currently selected set of sources \widehat{S}_s . The auxiliary datastructure \widehat{S}_c , initially set to \emptyset , is used to restrict the source overlaps considered by the *residualCoverage* algorithm. In each invocation, the algorithm first looks for the overlap statistics for $\{s\} \cup \widehat{S}_c$. If this statistic is among the learned (stored) statistics, the algorithm recursively invokes itself on supersets of $\{s\} \cup \widehat{S}_c$. Otherwise, the recursion stops in that branch (eliminating all the redundant superset lookups).

7 Experimental Setting

We now describe the data, algorithms and metrics of our experimental evaluation.

7.1 Database Set

To evaluate our approach, we use two kinds of databases: "Controlled" databases that we assembled locally and that allowed us to control the levels of correlations among those data sources, and the real "Web" data sources integrated by *BibFinder*.

Controlled Data Set: We designed a simple mediator which only exports data for the paper relation (see the Motivating Example). In Figure 13, we show the two hand-coded AV hierarchies we

CID	Source	Count
1	(S_2, S_7)	1.0
2	(S_1, S_2, S_3)	0.38
2	(S_1, S_2)	0.2
2	S_3	0.09

Table 1. Tuples in the table sourceInfo

use. We setup 20 data sources each of which contains data for the global relation *paper*. The data of the sources are the papers in DBLP published by computer science researchers in Database and Artificial Intelligence. The sources have different concentration of the data. For example, one source may contain only papers published in SIGMOD after 1996. Some of the sources are highly correlated for some queries. The queries can be selection query with conference and/or year attribute bound. The query list we used to discover frequent query classes was generated manually. The frequency of the queries are assigned proportionally to the statistics provided by NEC Research Index. We combine the statistics about the impact of the conferences and the most frequently accessed papers in Research Index to simulate the query frequency of our experimental system. We setup a one second delay for answering each query sent to a source to simulate the probing cost. Since there are no real users, we have to probe the sources to get the coverage and overlap statistics for the queries in the query list.

In the *BibFinder* scenario, we record and analyze these results as the queries are being asked, so no probing is needed there. However for some mediator without a maintained query list, probing sources could be very costly. So we also evaluated our proposed probing strategies using our controlled data set. In order to reduce the probing cost while still maintaining reasonable overall accuracy, for each class we only probe the sources using a small number of probing queries with high query probabilities. The probing queries are chosen in the descending order of their query probability from queries for the class and in *QList*. We use a threshold *minprobe* to decide when to stop probing. The threshold *minprobe* controls the minimum percentage of the total query frequency of all the queries in the class covered by the chosen probing queries. For example, for the class *ICDE&RT* in example 2, since query *ICDE&2001* covers 83.3% of the total query frequency of the class, while query *ICDE&RT* only covers 16.7%. So we will select query *ICDE&2001* as the first probing query, and we will not select query *ICDE&RT* as probing query until the threshold *minprobe* is larger than 83.3%. The probing results of the class will be stored in a table **sourceInfo**(CID, Source, Count), which keeps track of which answer tuples for the probing queries come from which sources. Each of the discovered frequent classes are given a class identifier, CID. For each class, we remember the weighted sum $\sum_Q P(Q) * \frac{N'_Q(\widehat{S})}{N_Q}$ for each source set \widehat{S} with at least one common answer for all the probing queries Q in the class. Here $N'_Q(\widehat{S})$ is the number of common tuples from the sources in the source set \widehat{S} for the query Q which are not reported for any

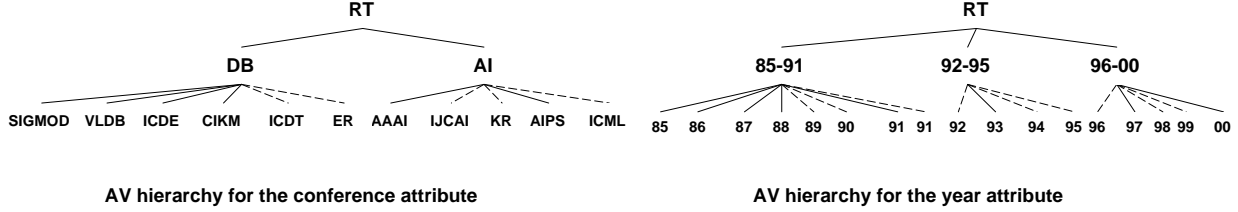


Figure 13. AV Hierarchies

superset of \hat{S} . Since the probing queries may be subsumed by multiple frequent query classes, we keep the results for each of these classes in the *sourceInfo* table too to avoid re-probing. A class will not have any entry in the *sourceInfo* table until we have some probing results for that class. In table 1, we give an example *sourceInfo* table.

Real Data Set in BibFinder : 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. 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, the Title hierarchy keeps only 1200 frequently asked keyword itemsets, the Conference hierarchy has more than 600 distinct values, and the Year hierarchy has 95 distinct values. Note that we consider a range query (for example: ">1990") as a single distinct value.

7.2 Algorithms:

In order to evaluate the effectiveness of our learned statistics, we implemented the **Simple Greedy** and **Greedy Select** algorithms described in [FKL97] to generate query plans using the learned source coverage and overlap statistics. A simple **Random Select** algorithm is also used to randomly choose k sources as the top k sources.

7.3 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 the plan is evaluated by calling the sources in the plan and 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. Let *TopK* refer to the real top k sources, and *Selected(p)* refer to the k sources selected in the plan p . Then the *precision* of the plan p is:

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

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

statistics.

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

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

where \hat{S}_i denotes the i^{th} source set of all possible source sets in the mediator, $P'(\hat{S}_i|Q)$ denotes the estimated overlap (or coverage) of the source set \hat{S}_i for query Q , and *TestQuerySet* refers to the set of all test queries.

8 Experimental Results

8.1 Results over the Controlled Data Set

In Figure 14(a), we observe the number of candidate frequent query classes and the number of frequent query classes. As we can see from the figure, as we increase the threshold *minfreq*, the number of candidate frequent classes and frequent classes will both decrease, and there is a sharp drop for the small thresholds. We also see, for almost all the *minfreq* thresholds, we always prune more than a half of the candidate frequent class discovered from DFC with low class access probability.

In Figure 14(b), we observe the statistics learning time which includes the time for discovering frequent query classes, probing the sources and computing the coverage and overlap statistics. As you can see as we increase the *minfreq*, the total learning time decreases. In the experiment, we just probe the sources with a very small number of queries whose total frequency covers *minprobe*=20% of the total frequency of all the queries in the class. The threshold *minoverlap* is set to 0.5%.

In Figure 14(c), we observe the average number of answers by executing the plans generated by the three algorithms for the 100 randomly generated queries. In Figure 14(d), we observe the average precision of the plans. As we can see the plans generated using our learned statistics are much better both in terms of the number of answers we get and in terms of the precision of the plans for these queries than the ones generated without using any statistics.

Altogether the experiments show that our association rule mining approach can effectively control the number of statistics required by a mediator to deal with the tradeoff between the accuracy of the statistics and the cost of leaning and remembering these

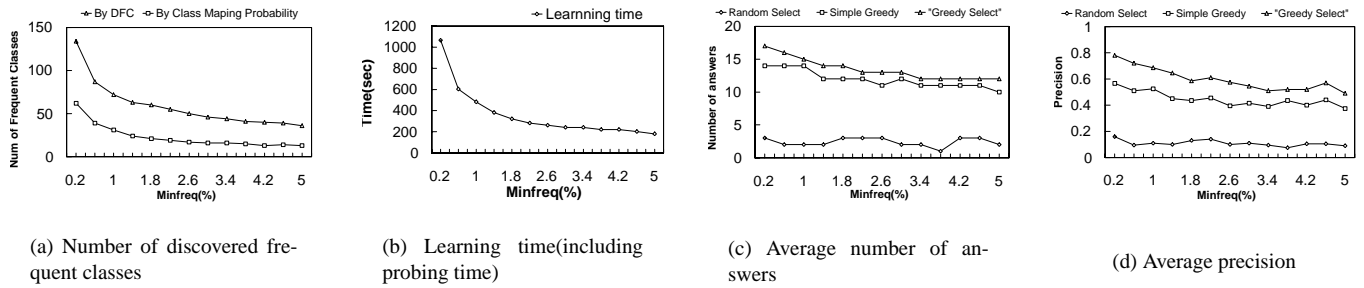


Figure 14. Experiment results

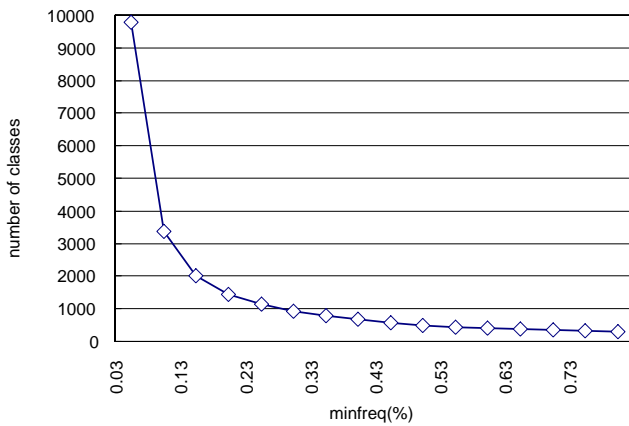


Figure 15. The total number of classes learned

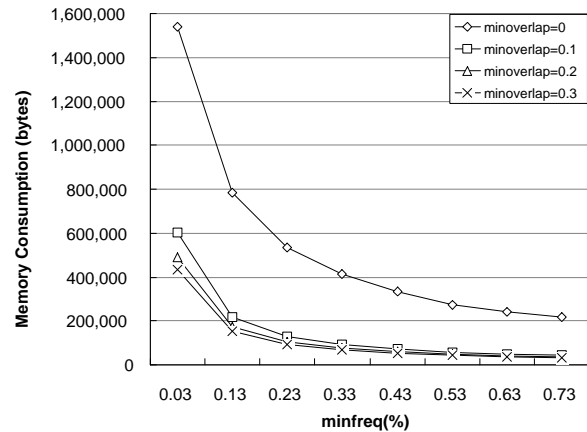


Figure 16. The total amount of memory needed for keeping the learned statistics in BibFinder

statistics. As we can see, the number of statistics and the learning time drop dramatically as we increase the threshold *minfreq*, while the average accuracy of the learned statistics drops smoothly.

8.2 Results over BibFinder

Space Consumption for Different *minfreq* and *minoverlap* Thresholds:

In Figure 15 and Figure 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 memory requirement 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 drops simultaneously. 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 ob-

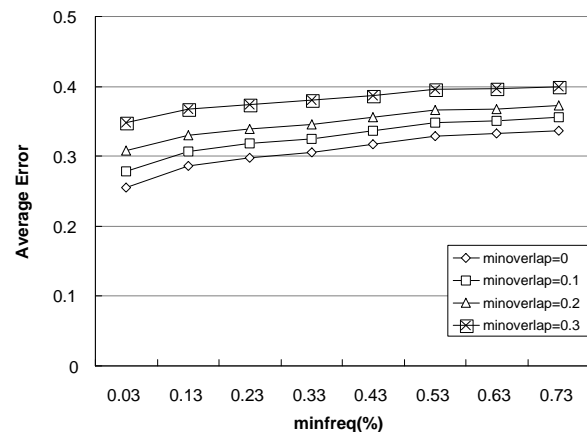


Figure 17. The average distance between the estimated statistics and the real coverage and overlap values.

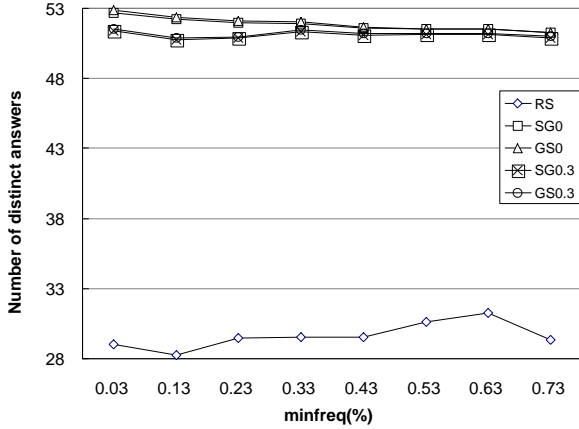


Figure 18. The average number of answers *BibFinder* returns by executing the query plans with top 2 sources.

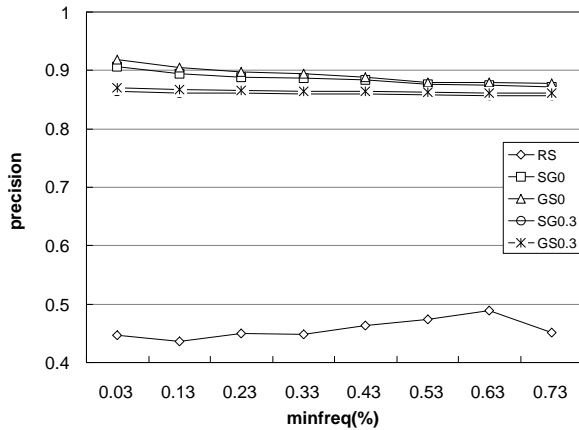


Figure 19. Precision for query plans with top 2 sources.

tained 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), memory requirement could become much larger as the number of users and the number of integrated sources grow.

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 Figure 16 and Figure 17 together, we can see that

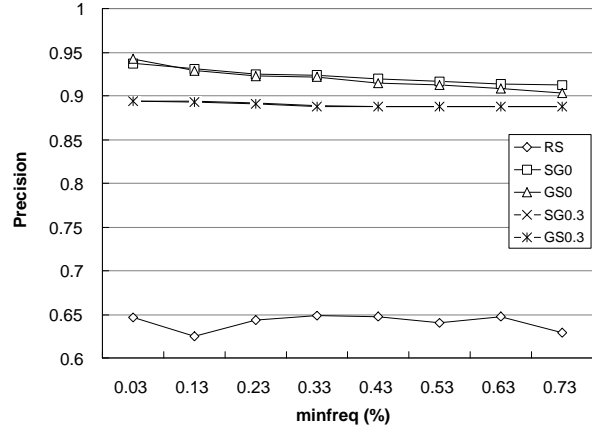


Figure 20. Precision for query plans with top 3 sources.

the absolute error of threshold combination: *minfreq* = 0.13% and *minoverlap* = 0.1 is almost the same as that of *minfreq* = 0.33% and *minoverlap* = 0, while the former uses only 50% of the memory required by the latter. This fact tells us that keeping very detailed overlap statistics of uncorrelated source sets for general query classes would not necessarily increase the accuracy of our statistics while requiring much more space.

Effectiveness of the Learned Statistics: We test 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, 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 Figure 19 and Figure 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 it decreases very slowly as we change the *minfreq* and *minoverlap* thresholds. Another 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

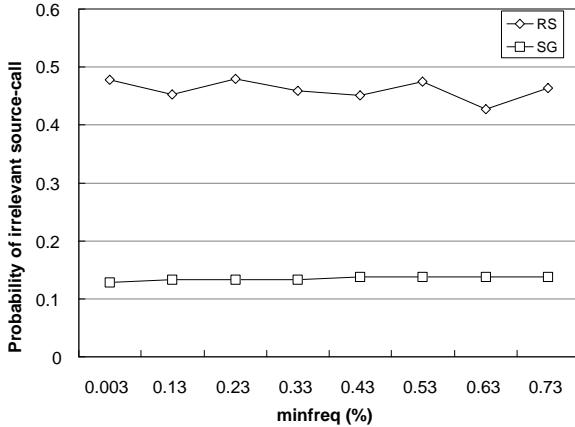


Figure 21. The percent of the total source-calls that are irrelevant for query plans with top 1 sources.

uses both coverage and overlap statistics. When we studied many queries asked by the *BibFinder* users and the corresponding coverage and overlap statistics, we found that the 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.

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 reduce the unnecessary load on its integrated sources.

Efficiency Issues: We now discuss the time needed for learning and using the coverage and overlap statistics. From the experiments, we found that using the learned statistics to generate query plans for a new query is very fast (i.e. always less than 1 millisecond). Additionally discovering frequent query classes and learning statistics is also fairly inexpensive (i.e. always less than 100 seconds). The most expensive phase is learning the AV Hierarchies. During the experiments we found that our GAVH algorithm can be very time-consuming when the number of attribute values gets larger. Specifically, it takes us 719ms to learn the Year hierarchy, 1 minute to learn the Conference hierarchy, 25 minutes to learn the Title keywords hierarchy, and 2.5 hours to learn the Author hierarchy⁶. However since GAVH runs offline and only needs to run

⁶Since the number of attribute values for the Title attribute is 8000, 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

once, it still is not a major drawback. Since it is the most time consuming phase, we can consider incrementally updating the hierarchy as new queries come in (see Section 9).

9 Discussion and Related Work

In our discussion we assume that queries asked by the users in future will have the same distribution as the past queries. Since the users' interests may change over different time periods, an important extension is to incrementally update the learned statistics w.r.t. the users' most recent interests. We are currently considering an incremental statistics updating approach to incrementally update AV Hierarchies and modify the existing class hierarchy by splitting, merging and deleting existing classes (and their respective statistics) in the class hierarchy.

In this paper, we only discussed how to learn coverage and overlap statistics of Select and Project queries. We concentrate on Select queries as those are the sort of queries asked in *BibFinder*, and use a single universal relation ([Nau02]) as it is a well-known assumption in Web data integration scenarios. The techniques described in this paper can however be extended to join queries. Specifically, we consider the join queries with the same subgoal relations together. For the join queries with the same subgoal relations, we can classify them based on their bound values and use similar techniques for selection queries to learn statistics for frequent join query classes.

Another assumption we have made was to assume the mediators will maintain a query list *QList*. However the *QList* may not be available for mediators at their beginning stages, the paper [NNVK02] introduces a size-based approach to learning statistics in such beginning scenarios. [NNVK02] assumes that query classes with more answer tuples will be accessed more frequently, and learns coverage statistics w.r.t. large query classes. Although the size-based approach can be seen as complementary to the frequency-based approach introduced in this paper, it's worth mentioning that the underlying technical details of the approaches are significantly different.

There has been some previous work on using probing techniques to learn database statistics both in multi-database literature and data integration literature. Zhu and Larson [ZL96] describe techniques for developing regression cost models for multi-database systems by selective querying. Adali et. al [ACPS96] discuss how keeping track of rudimentary access statistics can help in doing cost-based optimizations. More recently, the work by Gruser et. al. [GRZ⁺00] considers mining response time statistics for sources in data integration scenario. Given that both coverage and response time statistics are important for query optimization (c.f. [NK01,DH02]), our work can be seen as complementary to theirs.

The utility of quantitative coverage statistics in ranking the sources was first explored by Florescu et. al. [FKL97]. The primary aim of both these efforts was however on the "use" of coverage statistics, and they do not discuss how such coverage statistics

($O(n)$ complexity), and use GAVH to generate a hierarchy for these 2300 value clusters.

could be learned. In contrast, our main aim in this paper is to provide a framework for learning the required statistics.

There has also been some work on ranking text databases in the context of keyword queries submitted to meta-search engines. Recent work ([WMY00], [IGS01]) considers the problem of classifying text databases into a topic hierarchy. While our approach is similar to these approaches in terms of using concept hierarchies, and using probing and counting methods, it differs in several significant ways. The text database work uses a single topic hierarchy and does not have to deal with computation of overlap statistics. In contrast we deal with classes made up from the cartesian product of multiple AV hierarchies, and are also interested in overlap statistics. 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.

10 Conclusions

In this paper we motivated the need for automatically mining the coverage and overlap statistics of sources w.r.t. frequently accessed query classes for efficient query processing in a data integration scenario. We then presented a set of connected techniques that automatically generate AV Hierarchies, efficiently discover frequent query classes and learn coverage and overlap statistics for only these frequent classes. We described the details and implementation of our approach. We also presented an empirical evaluation of the effectiveness of our approach in BibFinder, a publicly available bibliography mediator. Our experiments demonstrate that (i) We can systematically trade the statistics learning time and number of statistics remembered for accuracy by varying the frequent class thresholds. (ii) The learned statistics provide tangible improvements in the source ranking, and the improvement is proportional to the type (coverage alone vs. coverage and overlap) and granularity of the learned statistics.

References

- [ACPS96] S. Adali, K. Candan, Y. Papakonstantinou, and V. S. Subrahmanian. Query caching and optimization in distributed mediator systems. In *Proceedings of SIGMOD-96*, 1996.
- [AS94] Rakesh Agrawal, Ramakrishnan Srikant. Fast Algorithms for Mining Association Rules. In *VLDB*, Santiago, Chile, 1994.
- [DH02] A. Doan and A. HaLevy. Efficiently Ordering Plans for Data Integration. In *Proceedings of ICDE-2002*, 2002.
- [DGL00] Oliver M. Duschka, Michael R. Genesereth, Alon Y. Levy. Recursive Query Plans for Data Integration. In *Journal of Logic Programming, Volume 43(1)*, pages 49-73, 2000.
- [FKL97] D. Florescu, D. Koller, and A. Levy. Using probabilistic information in data integration. In *Proceeding of the International Conference on Very Large Data Bases (VLDB)*, 1997.
- [GRZ⁺00] Jean-Robert Gruser, Louiqa Raschid, Vladimir Zadorozhny, Tao Zhan: Learning Response Time for WebSources Using Query Feedback and Application in Query Optimization. *VLDB Journal* 9(1): 18-37 (2000)
- [HK00] Jiawei Han and Micheline Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufman Publishers, 2000.
- [IGS01] P. Ipeirotis, L. Gravano, M. Sahami. Probe, Count, and Classify: Categorizing Hidden Web Databases. In *Proceedings of SIGMOD-01*, 2001.
- [LKG99] E. Lambrecht, S. Kambhampati and S. Gnanaprakasam. Optimizing recursive information gathering plans. In *Proceeding of the International Joint Conference on Artificial Intelligence (IJCAI)*, 1999.
- [LRO96] A. Levy, A. Rajaraman, J. Ordille. Query Heterogeneous Information Sources Using Source Descriptions. In *VLDB Conference*, 1996.
- [Nau02] F. Naumann. *Quality-Driven Query Answering for Integrated Information Systems*. Volume 2261 of LNCS, Springer Verlag, Heidelberg, 2002.
- [NLF99] F. Naumann, U. Leser, J. Freytag. Quality-driven Integration of Heterogeneous Information Systems. In *VLDB Conference 1999*.
- [NK01] Z. Nie and S. Kambhampati. Joint optimization of cost and coverage of query plans in data integration. In *ACM CIKM*, Atlanta, Georgia, November 2001.
- [NKH03] Z. Nie, S. Kambhampati and T. Hernandez. BibFinder/StatMiner: Effectively Mining and Using Coverage and Overlap Statistics in Data Integration. In *Proceedings of VLDB 2003*.
- [NNVK02] Z. Nie, U. Nambiar, S. Vaddi and S. Kambhampati. Mining Coverage Statistics for Webservice Selection in a Mediator. *Proc. CIKM 2002*.
- [PL00] Rachel Pottinger, Alon Y. Levy, A Scalable Algorithm for Answering Queries Using Views. *Proc. of the Int. Conf. on Very Large Data Bases (VLDB) 2000*.
- [WMY00] W. Wang, W. Meng, and C. Yu. Concept Hierarchy based text database categorization in a metasearch engine environment. In *WISE2000*, June 2000.
- [ZL96] Q. Zhu and P-A. Larson. Developing Regression Cost Models for Multi-database Systems. In *Proceedings of PDIS*. 1996.