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

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 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 fielded 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 [1, 4, 9, 10, 14, 16] 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 contains a partial extension of the relation. Query optimization in data integration [3, 5, 11, 12] 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*.

*This research is supported in part by the NSF grant IRI-9801676 and the ASU ET-I^3 initiative grant ECR A601. We thank Louiqa Raschid, Huan Liu, K. Selcuk Candan, Ullas Nambiar, and especially Thomas Hernandez for many helpful critiques.

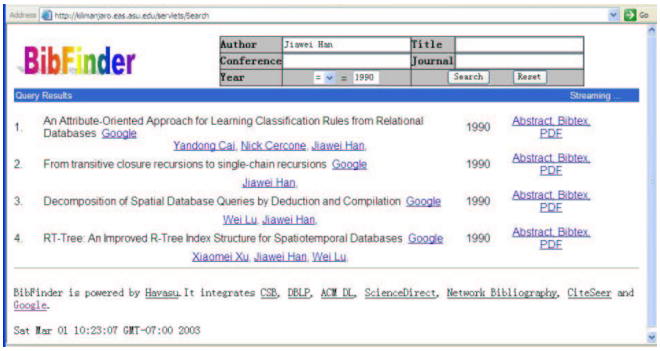


Figure 1. The BibFinder User Interface

BibFinder Scenario: We have been developing *BibFinder* (Figure 1, 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, ACM Guide, IEEE Xplore, ScienceDirect,* and *CiteSeer.* Plans are underway to add several additional sources including *AMS MathSciNet* and *Computational Geometry Bibliography*. Since its unveiling in December 2002, *BibFinder* has been getting on the order of 200 queries a day.

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

Need for Statistics: To efficiently answer user queries, it is im-

portant to find and access the most relevant subset of the sources for the given query. Suppose the user asks a selection query

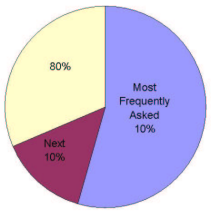
Q(title,author,year) :−
paper(title, author, conference/journal, year),
conference/journal ="SIGMOD".

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]

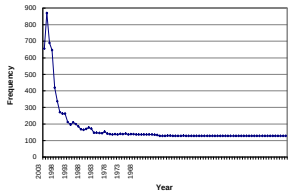
Coverage and Overlap Statistics: In order to judge the source relevance however, *BibFinder* needs to know the *coverage* of each source S with respect to the query Q, i.e. $P(S|Q)$, the probability that a random answer tuple for query Q belongs to source S. Given this information, we can rank all the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

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 learning and storage costs down is to learn statistics only with respect to a smaller set of "frequently asked" queries that are likely to be most useful in answering user queries. This strategy trades accuracy of statistics for reduced statistics learning/storing costs. In the *BibFinder* scenario, for example, we could learn statistics with respect to the list of queries that are actually posed to the mediator over a period of time. The motivation for such an approach is that even if a mediator cannot provide accurate statistics for every possible query, it can still achieve a reasonable average accuracy by keeping more accurate coverage and overlap statistics for queries that are asked more frequently, and less accurate statistics for infrequent queries. The effectiveness of this approach is predicated on the belief that in most real-world scenarios, the distribution of queries posed to a mediator is not *uniform*, but rather Zipfian. This belief is amply validated in *BibFinder*. Figure 2 shows the distribution of the keywords, and bindings for the Year attribute used in the first 15000 queries that were posed to *BibFinder*. Figure 2(a) shows that the most frequently asked 10% keywords appear in almost 60% of all the selection queries binding attribute Title. Figure 2(b) shows that the users are much more interested in recently published papers.



(a) Keywords Distribution



(b) Queries binding attribute year

Figure 2. 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.

[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.

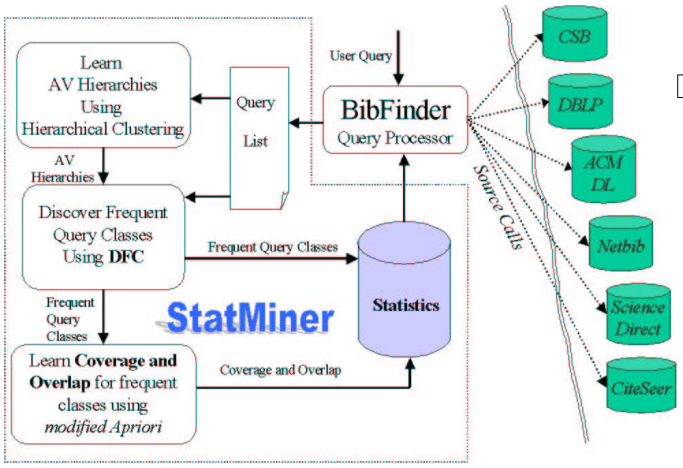


Figure 3. StatMiner Architecture

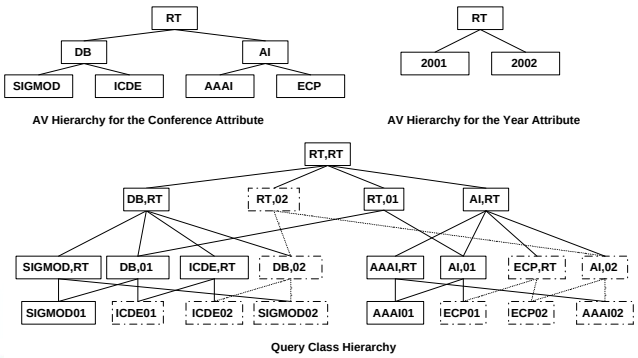


Figure 4. AV Hierarchies and the Corresponding Query Class Hierarchy

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). 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*. Figure 3 shows the high-level architecture of *StatMiner*. *StatMiner* starts with a list of workload queries. The query list is collected from the log of queries submitted to *BibFinder*, and not only gives the specific queries submitted to *BibFinder*, but also coverage and overlap statistics on how many tuples for each query came from which source. The query list is used to automatically learn AV hierarchies, and to prune query classes that subsume less than a given number of user queries (specified by a frequency threshold). For each of these remaining classes, class-source as well as class-source set association rules are learned. An example of a class-source association rule could be that $SIGMOD \rightarrow DBLP$ with confidence 100%, which means that the information source $DBLP$ covers all the paper information for $SIGMOD$ related queries.

[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!

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. 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. 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 Section 10 presents our conclusions.

2 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 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 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 3, we give details on how we learn them automatically.

Query Classes: Since a typical selection query will have values

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

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 i^{th} classificatory attribute. Then the set of all query classes (called $classSet$) is simply $H_1 \times H_2 \times ... \times H_n$. The AV hierarchies induce subsumption relations among the query classes. A class C_i is subsumed by class C_j if every feature in C_i is equal to, or a specialization of, the same dimension feature in C_j. A query Q is said to belong to a class C if the values of the classificatory attributes in Q are equal to, or are specializations of, the features defining C. Figure 4 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 5, 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 are measured relative to the available sources.

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

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

The coverage and overlap statistics w.r.t. a class C is used to estimate the source coverage and overlap for all the queries that are mapped into C. These statistics can be conveniently computed using an association rule mining approach as discussed below.

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

3 Generating AV Hierarchies Automatically

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

Distance Function: The main idea of generating an AV hierarchy is to cluster similar attribute values into classes in terms of

the coverage and overlap statistics of their corresponding selection queries binding these values. The problem of finding similar attribute values becomes the problem of finding similar selection queries. In order to find similar queries, we define a distance function to measure the distance between a pair of selection queries (Q_1, Q_2):

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

Where $\widehat{S}_i$ denotes the i^{th} source set of all possible source sets in the mediator. Although the number of all possible source sets is exponential in terms of the number of available sources, we only need to consider source sets with answers for at least one of the two queries to compute $d(Q_1, Q_2)$.[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. The statistics $P(\hat{S}|Q)$ for a specific query Q are computed using the statistics from the query list maintained by the mediator.

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 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.

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

[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.

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

Figure 6. The GAVH algorithm

and overlap statistics vector $\overrightarrow{P(\widehat{S}|C)}$ by using the feature vectors of its children clusters C_1, C_2:

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

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

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

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

general queries we use the statistics of the specific queries[4] within these classes using the following formula:

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

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 queries within the class.

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 with such kinds of AV Hierarchies is that the level of abstraction may not actually increase when we go up the hierarchy. For example, in Figure 7, assuming the three attribute values have the same coverage and overlap statistics, then we should not put them into separate clusters. If we put these attribute values into two clusters C_1 and C_2, these two clusters are essentially in the same level of abstraction. Therefore we may waste our memory space on remembering the same statistics multiple times.

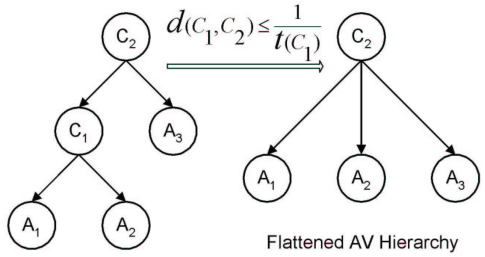


Figure 7. An example of Flattening AV Hierarchy

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 entire hierarchy.

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

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

Figure 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.

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 amount of statistics low, we would like to prune query classes which are rarely accessed. In this section we describe how frequently accessed classes are discovered in a two-stage process.

We use the term *candidate frequent class* to denote any class with class probability greater than the minimum frequency threshold *minfreq*. The example classes shown in Figure 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 are not subsumed by each other. For example, the query (or class) (ICDE,01) has both the class (DB,01) and class (ICDE,RT) as its parent class. For a query with multiple ancestor classes, we need to map the query into a set of least-general ancestor classes which are not subsumed by each other (see Section 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 is estimated using the following formula:

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

Since the class access probability of a candidate frequent class will be affected by the distribution of other candidate frequent classes, in order to identify the classes with high class access probability, we have to discover all the candidate frequent classes first. In the next subsection, we will introduce an algorithm to discover candidate frequent classes. In Section 4.2, we will then discuss

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

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

Figure 9. The DFC algorithm

Procedure *genClassSet(k : number of features;Q : the query; $classSet$: discovered frequent class set; AV hierarchies)*

for *(each feature $f_i \in Q$)*
$ftSet_i = \{f_i\}$;
$ftSet_i = ftSet_i \cup (\{ancestor(f_i)\} - \{root\})$;
end for
candidateSet={};
for *(each k feature combination $(ftSet_{j_1}, ..., ftSet_{j_k})$)*
$tempSet = ftSet_{j_1}$;
for *($i = 1; i < k; i++$)*
remove any class $C \notin classSet_i$ from tempSet;
$tempSet = tempSet \times ftSet_{j_{i+1}}$;
end for
remove any class $C \notin classSet_{k-1}$ from tempSet;
$candidateSet = candidateSet \cup tempSet$;
end for
return $candidateSet$;
End *genClassSet*;

Figure 10. Ancestor class set generation procedure

how to prune candidate frequent classes with low class access probability.

4.1 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] ([7]) 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 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 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.

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

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,

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

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 $minfreq$.

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

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 (see Section 6), and update the corresponding statistics vectors $\overrightarrow{P(\widehat{S}|C)}$ of its ancestor classes using the query's coverage and overlap statistics vector $\overrightarrow{P(\widehat{S}|Q)}$ through the formula $\overrightarrow{P(\widehat{S}|C)} = \frac{\sum_{Q \in C} \overrightarrow{P(\widehat{S}|Q)} \times P(Q)}{P(C)}$. When the mapping and updating procedure is completed, we simply need to prune the overlap statistics which are smaller than the threshold $minoverlap$. One potential problem of this naive approach is the possibility of running out of memory, since the system has to remember the coverage and overlap statistics for each source set and class combination. If the mediator has access to n sources and has discovered m frequent classes, then the memory requirement for learning these statistics is $m \times 2^n \times k$, where k is the number of bytes needed to store a float number. If $k = 1$, $m = 10000$, and the total number of memory available is $1GB$, this approach would not scale well when the number of sources is greater than 16.

In order to handle scenarios with large number of sources, we use a modified Apriori algorithm ([2]) to avoid considering any supersets of an uncorrelated source set. We first identify individual sources with coverage statistics greater than *minoverlap*, and keep coverage statistics for these sources. Then we discover all 2-$sourceSet$ [6] with overlap greater than *minoverlap*, and keep only overlap statistics for these source sets. This process continues until we have the overlap statistics for all the correlated source sets.

6 Using Learned Coverage and Overlap Statistics

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

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

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

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

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

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

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

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

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

Using Coverage and Overlap Statistics to Generate Query Plans: Once we have the coverage and overlap statistics, we use the **Simple Greedy** and **Greedy Select** algorithms described in [5] to generate query plans. Specifically, *Simple Greedy* generates plans by greedily selecting the top k sources ranked only according

[6] k-$sourceSet$ denotes the source sets with only k sources.

to their coverages, while *Greedy Select* selects sources with high residual coverages calculated using both the coverage and overlap statistics. A residual coverage computing algorithm is discussed in [13] 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 resulting efficiency is critical in practice since computation of residual coverage forms the inner loop of any query processing algorithm that considers source coverage.

7 Experimental Setting

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

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 25000 real queries asked by *BibFinder* users as of May 20, 2003 as the query list. Among them, we randomly chose 4500 queries as test queries and the others were used as training data. The AV Hierarchies for all four attributes were learned automatically using our GAVH algorithm. The learned Author hierarchy has more than 8000 distinct values,[7] 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.

Algorithms: In order to evaluate the effectiveness of our learned statistics, we implemented the **Simple Greedy** and **Greedy Select** algorithms to generate query plans using the learned source coverage and overlap statistics. A simple **Random Select** algorithm was 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 true top k sources after we execute the query.

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

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

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

[7] 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.

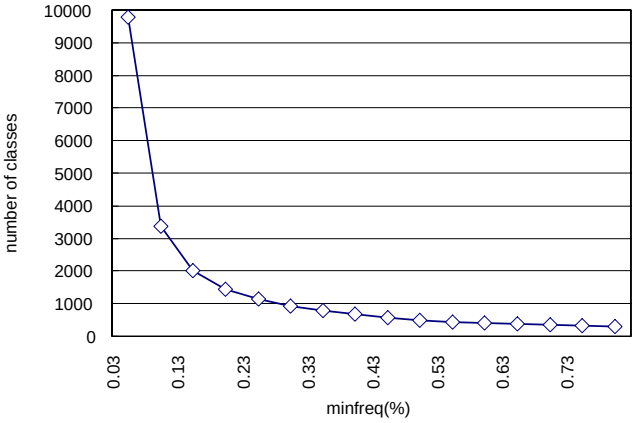


Figure 12. The total number of classes learned

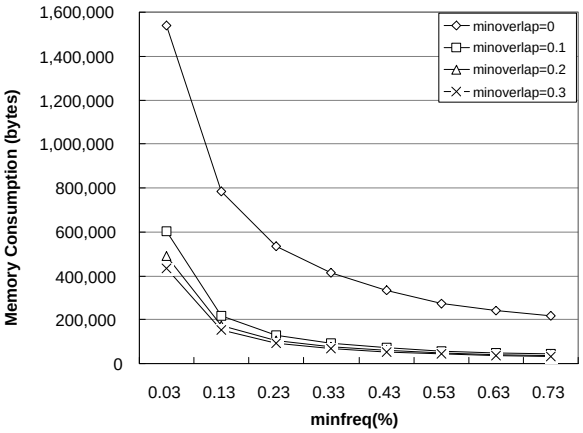


Figure 13. The total amount of memory needed for keeping the learned statistics in BibFinder

8 Experimental Results

Space Consumption for Different *minfreq* and *minoverlap* Thresholds: In Figures 12 and 13, we observe the reduction in space consumption and number of classes when we increase the

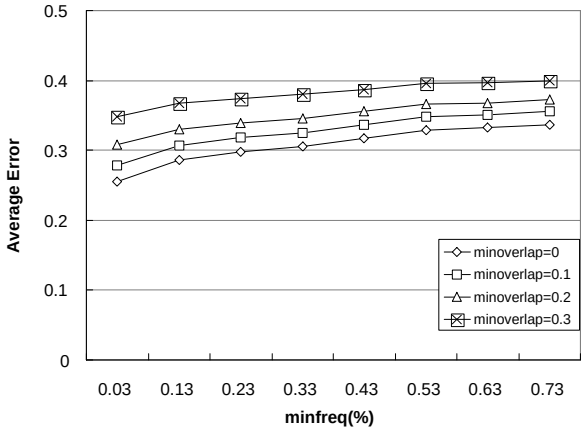


Figure 14. The average distance between the estimated statistics and the real coverage and overlap values.

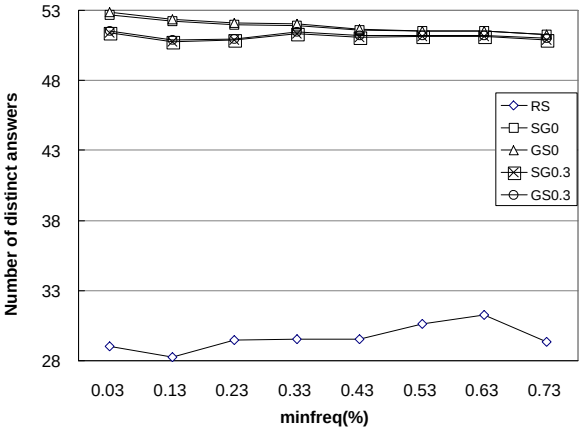


Figure 15. The average number of answers *BibFinder* returns by executing the query plans with top 2 sources.

minfreq and *minoverlap* thresholds. Slightly increasing the *minfreq* threshold from 0.03% to 0.13% causes the number of classes to drop dramatically from approximately 10000 classes to 3000. As we increase the *minfreq* threshold, the number of classes decreases, however the decrease rate becomes smaller as the threshold becomes larger. In Figure 13, 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.[8]

Accuracy of the Learned Statistics for Different *minfreq* and *minoverlap* Thresholds: Figure 14 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 13 and 14 together, we can see that the absolute error of threshold combination: $minfreq = 0.13\%$ and $minoverlap = 0.1$ is almost the same as that of $minfreq = 0.33\%$ and $minoverlap = 0$, while the former uses only 50% of the memory required by the latter. This fact tells us that keeping very detailed overlap statistics of uncorrelated source sets for general query classes would not necessarily increase the accuracy of our statistics while requiring much more space.

Effectiveness of the Learned Statistics: We evaluate the effectiveness of the learned statistics by actually testing these statistics

[8]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.

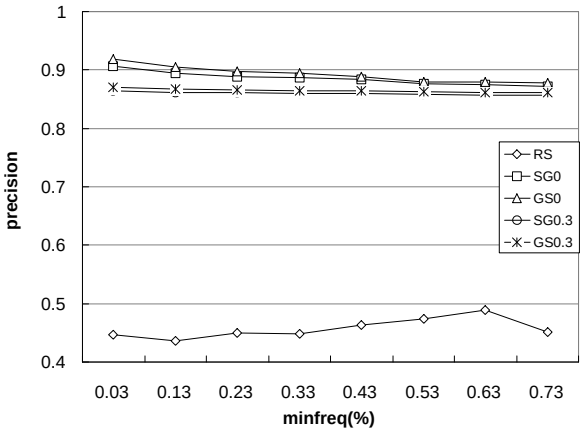


Figure 16. Precision for query plans with top 2 sources.

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 15, 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

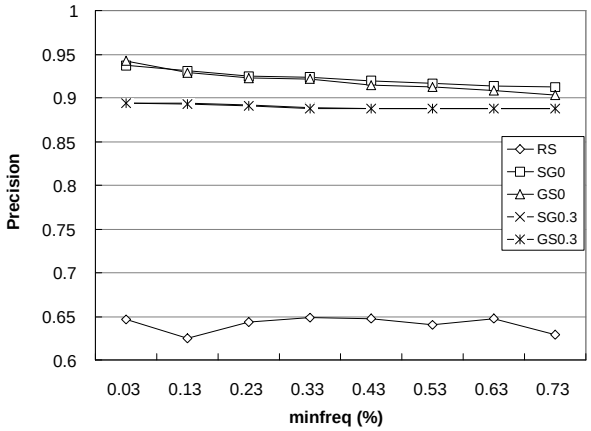


Figure 17. Precision for query plans with top 3 sources.

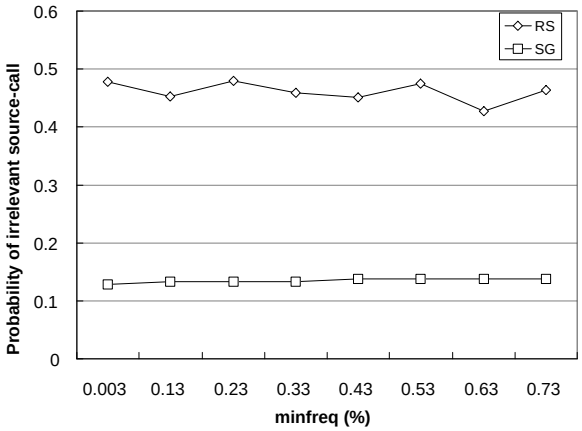


Figure 18. The percent of the total source-calls that are irrelevant for query plans with top 1 sources.

plans selected by Simple Greedy and Greedy Select, while we can only get about 30 distinct answers by randomly selecting 2 sources. In Figures 16 and 17, we observe the average precision of the top 2 and top 3 sources ranked using statistics with different level of abstraction for the test queries. As we can see, the plans using our learned statistics have high precision, and their precision decreases very slowly as we change the *minfreq* and *minoverlap* thresholds.

One fact we need to point out is that the precision of the plans using Simple Greedy and Greedy Select algorithm are very close (although Greedy Select is a little better most of the time). This is not as we expected, since the Simple Greedy only uses the coverage statistics, while Greedy Select uses both coverage and overlap statistics. When we studied many queries asked by the *BibFinder* users and the corresponding coverage and overlap statistics, we found that the distribution of answer tuples over sources integrated by *BibFinder* almost follow independence assumption for most of the queries asked by the users. However in other scenarios Greedy Select can perform considerably better than Simple Greedy. For instance, in our 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 [13].

Figure 18 shows the possibility of a source call being a completely irrelevant source call (i.e. the source has no answer for the query asked). The graph reveals that the most relevant source selected using our algorithm has only 12% possibility of being an irrelevant source call, while the randomly picked source has about 46% possibility. This illustrates that by using our statistics *BibFinder* can significantly reduce the unnecessary load on its integrated sources.

Efficiency Issues: We now discuss the time needed for learning and using the coverage and overlap statistics. All our experiments were run under JDK 1.2.2 on a 500MHZ SUN-Blade-100 with 768Mb of RAM. From the experiments, we found that using the learned statistics to generate query plans for a new query is very fast, specifically always less than 1 millisecond. In terms of the statistics learning, costs associated with discovering frequent query classes and learning statistics are also fairly inexpensive (i.e. always less than 100 seconds). Our previous experiments with 20 artificial sources (see [13]) also shows that our statistics learning algorithms can scale well. The most expensive phase is learning the AV Hierarchies. During the experiments we found that the GAVH algorithm can be very time-consuming when the number of attribute values is large. Specifically, it takes us 719ms to learn the Year hierarchy, 1 minute to learn the Conference hierarchy, 25 minutes to learn the Title keywords hierarchy, and 2.5 hours to learn the Author hierarchy. However since GAVH runs offline and only needs to run once, it still is not a major drawback. Since it is the most time consuming phase, we can consider incrementally updating the hierarchy as new queries come in.

9 Related Work

The utility of quantitative coverage statistics in ranking the sources was first explored by Florescu *et. al.* [5]. The primary aim of the effort was however on the "use" of coverage statistics, and it does not discuss how such coverage statistics could be learned. In contrast, our main aim in this paper is to provide a framework for *learning* the required statistics.

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 [18] describe techniques for developing regression cost models for multi-database systems by selective querying. Adali *et. al.* [1] discuss how keeping track of rudimentary access statistics can help in doing cost-based optimizations. More recently, the work by Gruser *et. al.* [6] considers mining response time statistics for sources in data integration scenario. In contrast,

our work focuses on learning coverage and overlap statistics. As has been argued by us [12] and others [3], query optimization in data integration scenarios require both types of statistics.

Another strand of related work [8, 17] considers the problem of text database selection in the context of keyword queries submitted to meta-search engines. Although some of these efforts use a hierarchy of topics to categorize the Web sources, they use only a single topic hierarchy and do not deal with computation of overlap statistics. In contrast we deal with classes made up from the cartesian product of multiple attribute value hierarchies, and are also interested in modeling overlap. This makes the issue of space consumed by the statistics quite critical for us, necessitating our threshold-based approaches for controlling the resolution of the statistics. Furthermore, most of the existing approaches in text database selection assume that the terms in a user's query are independent (to avoid storing too many statistics). No efficient approaches have been proposed to handle correlated keyword sets. We are currently working on applying our techniques to the text database selection problem to effectively solve the space and learning overhead brought by providing coverage and overlap statistics for both single word and correlated multi-word terms.

In the current work, we have assumed that the mediator will maintain a query list. However the query list may not be available for mediators at their beginning stages. For such cases our earlier work [15] introduces a size-based approach to learning statistics. There we assume that query classes with more answer tuples will be accessed more frequently, and learn coverage statistics with respect to large query classes.

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 attribute value hierarchies, efficiently discover frequent query classes and learn coverage and overlap statistics for only these frequent classes. We described the algorithmic details and implementation of our approach. We also presented an empirical evaluation of the effectiveness of our approach in *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 granularity of the learned statistics. A prototype of the *BibFinder* system using these statistics was demonstrated at *VLDB* 2003 [14].

One direction that we are currently pursuing is to make the statistics learning framework more adaptive to changes in user interests (as captured by the query list) as well as the number and content of the data sources. We believe that our general approach can be easily made incremental to support these goals.

References

[1] S. Adali, K. Candan, Y. Papakonstantinou, and V. S. Subrahmanian. Query caching and optimization in distributed mediator systems. In *Proc. of SIGMOD*, 1996.

[2] R. Agrawal and R. Srikant. Fast Algorithms for Mining Association Rules. In *Proc. of VLDB*, Santiage, Chile, 1994.

[3] A. Doan and A. Halevy. Efficiently Ordering Plans for Data Integration. In *Proc. of ICDE*, 2002.

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

[5] D. Florescu, D. Koller, and A. Levy. Using probabilistic information in data integration. In *Proc. of VLDB*, 1997.

[6] J. Gruser, L. Raschid, V. Zadorozhny, and T. Zhan: Learning Response Time for WebSources Using Query Feedback and Application in Query Optimization. In *VLDB Journal 9(1)*, pages 18-37, 2000.

[7] J. Han and M. Kamber. Data Mining: Concepts and Techniques. Morgan Kaufmman Publishers, 2000.

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

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

[10] A. Levy, A. Rajaraman, and J. Ordille. Query Heterogeneous Information Sources Using Source Descriptions. In *Proc. of VLDB*, 1996.

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

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

[13] Z. Nie and S. Kambhampati. A Frequency-based Approach for Mining Coverage Statistics in Data Integration. ASU CSE TR 03-004. Dept. of Computer Science & Engg. Arizona State University.
$http://www.public.asu.edu/\sim zaiqingn/freqbased.pdf$

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

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

[16] R. Pottinger and A. Y. Levy , A Scalable Algorithm for Answering Queries Using Views. In *Proc. of VLDB*, 2000.

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

[18] Q. Zhu and PA Larson. Developing Regression Cost Models for Multi-database Systems. In *Proc. of PDIS*, 1996.