Mining Coverage Statistics for Websource Selection in a Mediator

Zaiqing Nie Ullas Nambiar Sreelakshmi Vaddi Subbarao Kambhampati*

Department of Computer Science and Engineering

Arizona State University, Tempe AZ 85287-5406

{nie, mallu, slakshmi, rao}@asu.edu

ASU CSE TR 02-009

Abstract

Recent work in data integration has shown the importance of statistical information about the coverage and overlap of sources for efficient query processing. Despite this recognition there are no effective approaches for learning the needed statistics. The key challenge in learning such statistics is keeping the number of needed statistics low enough to have the storage and learning costs manageable. Naive approaches can become infeasible very quickly. In this paper we present a set of connected techniques that estimate the coverage and overlap statistics while keeping the needed statistics tightly under control. Our approach uses a hierarchical classification of the queries, and threshold based variants of familiar data mining techniques to dynamically decide the level of resolution at which to learn the statistics. We describe the details of our method, and present experimental results demonstrating the efficiency of the learning algorithms and the effectiveness of the learned statistics.

Keywords: *Webmining to support query optimization, Web-based Data Integration, Coverage Statistics*.

*Contact Author. Preliminary versions of this work have been presented at WIDM 2001 [NKNV01] and CIKM 2002 [NNVK02].

1 Introduction

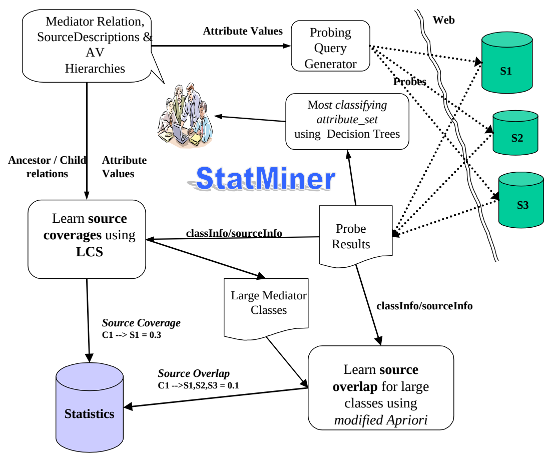
With the vast number of autonomous information sources available on the Internet today, users have access to a large variety of data sources. Data integration systems [CGHI94, LRO96, ACPS96, LKG99, PL00] are being developed to provide a uniform interface to a multitude of information sources, query the 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, DL99, NLF99, NK01] 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. We illustrate the need for these statistics with an example.

Example 1: Consider a *Simple Mediator* that integrates autonomous Internet sources exporting information about publications in Computer Science. Let there be a single relation in the global schema of the mediator: **paper(title, author, conference, year)**. There are hundreds of Internet sources, each exporting a subset of the global relation. Some sources may only contain publications in AI, others may focus on Databases, while some others on Bioinformatics etc. To efficiently answer users' queries, we need to find and access the most relevant subset of the sources for the given query.[1] Suppose, the user asks a selection query:

Q(title,author) :− **paper**(title, author, conference, year),
conference="AAAI".

To answer this query efficiently, we need 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. Although

[1]Although in the following example, as well as in the subsequent discussion the sources are seen as directly using the mediator schema, in practice, the sources may have different schemas, with varying relation as well as attribute names. We ignore this issue in our discussion as all data integration frameworks assume that the mapping between the source and mediator ontology is supplied by the source-mediator relations [DGL00,LRO96].



Mediator Relation, SourceDescriptions & AV Hierarchies
Attribute Values
Probing Query Generator
Web
S1
Probes
S2
Most classifying attribute_set using Decision Trees
Ancestor / Child relations
Attribute Values
StatMiner
S3
Learn source coverages using LCS
classInfo/sourceInfo
Probe Results
Large Mediator Classes
classInfo/sourceInfo
Source Coverage C1 --> S1 = 0.3
Source Overlap C1 -->S1,S2,S3 = 0.1
Statistics
Learn source overlap for large classes using modified Apriori

Figure 1: *StatMiner Architecture*

ranking seems to provide the complete order in which to access the sources, this is unfortunately not true in general. It is quite possible that the two sources with the highest coverage with respect to Q happen to mirror each others' contents. Clearly, calling both sources is not going to give any more information than calling just one source. Therefore, after we access the source S' with the maximum coverage $P(S'|Q)$, we need to access as the second source, the source S'' that has 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 pick the source S'' that has next best rank to S' in terms of $P(S|Q)$ but has minimal *overlap* (common tuples) with S'.

Given that sources tend to be autonomous in a Web data integration scenario and that the mediation may or may not be authorized, it is impractical to assume that the sources will automatically export coverage and overlap statistics. Consequently, Web data integration systems should be able to learn the necessary statistics. Although previous work has addressed the issue of how to model these statistics (c.f. [FKL97]), and how to *use* them as part of query optimization (c.f. [DL99,NLF99,NK01]), there has not been any work on effectively learning the statistics in the first place.

1.1 The *StatMiner* approach

In this paper, we address the problem of learning the coverage and overlap statistics for sources with respect to user queries. A naive approach may involve learning the coverages and overlaps of all sources with respect to all queries. This will necessitate $N_q * 2^{N_S}$ different statistics, where N_q is the number of different queries that the mediator needs to handle and N_S is the number of data sources that are integrated by the mediator. An important challenge is to keep the number of statistics under control, while still retaining their advantages.

In this paper, we present *StatMiner* (see Figure 1), a statistics mining module for web based data integration. *StatMiner* is being developed as part of the *Havasu* data integration project at Arizona State University [KNNV02]. *StatMiner* comprises of a set of connected techniques that estimate the coverage and overlap statistics while keeping the amount of needed statistics tightly under control. Since the number of potential user queries can be quite high, *StatMiner* aims to learn the required statistics for *query classes* i.e. groups of queries. A *query class* is an instantiated subset of the global relation and contains only the attributes for which a hierarchical classification of instances (values) can be generated. Thus for the global relation *paper* in Example 1, *StatMiner* may generate query classes using the attributes $conference$ and $year$. By selectively deciding the level of generality of the query classes with respect to which the coverage statistics are learnt, *StatMiner* can tightly control the number of needed statistics (at the expense of loss of accuracy). The loss of accuracy may not be a critical issue for us as it is the *relative* rather than the *absolute* values of the coverage statistics that are more important in ranking the sources.

The coverage statistics learning is done using the LCS algorithm, and the overlap statistics using a variant of the Apriori algorithm [AS94]. LCS algorithm does two things: it identifies the query classes which have large enough support, and it computes the coverages of the individual sources with respect to these identified large classes. The resolution of the learned statistics is controlled in an adaptive manner with the help of two thresholds. The threshold τ_c is used to decide whether a query class has large enough support to be remembered. When a particular query class doesn't satisfy the minimum support threshold, *StatMiner*, in effect, stores statistics only with respect to some abstraction (generalization) of that class. Another threshold τ_o is used to decide whether or not the overlap statistics between a set of sources and a remembered query class should be stored.

Specifically, *StatMiner* probes the Web sources exporting the mediator relation. Using LCS we then classify the results obtained into the query classes and dynamically identify "large" classes for which the number of results mapped are above the specified threshold τ_c. We learn and store statistics only w.r.t. these identified large classes. When the mediator, in our case *Havasu*, encouters a new user query, it maps the query to one of the query classes for which statistics are available. Since we use thresholds to control the set of query classes for which statistics are maintained, it is possible that there is no query class that exactly matches the user query. In this case, we map the query to the nearest abstract query class that has available statistics. The loss of accuracy in statistics entailed by this step should be seen as the cost we pay for keeping the amount of stored statistics low. Once the query class corresponding to the user query is determined, the mediator uses the learned coverage and overlap statistics to rank the data sources that are most relevant to answering the query.

In order to make this approach practical, we need to carefully control three types of costs: (1) cost of getting the training data from the sources (i.e., "probing costs") (2) the cost of processing the data to compute coverage and overlap statistics (i.e., "mining costs") and (3) the online cost of *using* the coverage and overlap statistics to rank sources. In the rest of the paper, we shall explain how we control these costs. Briefly, the probing costs are controlled through sampling techniques. The "mining costs" are controlled with the help of support and overlap thresholds. The "usage costs" are controlled with the help of an efficient algorithm for computing the residual coverage. We will demonstrate the effectiveness of these mechanisms through empirical studies.

Although we focus purely on source selection order and thus, in effect, consider cost models based purely on coverage, the learned statistics can also be used in cost models that are based both on coverage and response time [NLF99,DL99,NK01]. Indeed the work on *StatMiner* was motivated by our earlier work([NK01]) on joint optimization of cost and coverage for query plans in data integration. In that work, we develop elaborate cost models for comparing query plans given a variety of cost/coverage tradeoffs. These cost models are, for example, able to handle the tradeoffs presented by high-coverage sources with slow response times vs. low-coverage sources with fast response times. The earlier work assumed the existence of coverage-related statistics, while in this work we explicitly consider how to learn them.

The rest of the paper is organized as follows. In the next section, we discuss the related work. Section 3 gives an overview of *StatMiner*. Section 4 describes the methodology used for extracting and processing training data from autonomous Web sources. In Section 5, we give the algorithms for learning coverage and overlap statistics. Then, in Section 6, we discuss how to efficiently use the leaned statistics to rank the sources for a given query. This is followed, in Section 7, by a detailed description of our experimental setup, and in Section 8, by the results we obtained demonstrating the efficiency of our learning algorithms and the effectiveness of the learned statistics. Section 9 contains a discussion of some practical issues regarding the realization of the *StatMiner* approach. We present our conclusions in Section 10.

2 Related Work

Researchers in data integration have long noted the difficulty of obtaining relevant source statistics for use in query optimization. There have broadly been two approaches for dealing with this difficulty. Some approaches, such as those in [LRO96,DGL00,LKG99] develop heuristic query optimization methods that either do not use any statistics, or can get by with very coarse statistics about the sources. Others, such as [NLF99,DL99,NK01], develop optimization approaches that are fully statistics (cost) based. While these approaches assume a variety of coverage and response-time statistics, they do not however address the issue of learning the statistics in the first place– which is the main focus of the current paper.

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. [DL99,NK01]), our work can be seen as complementary to theirs.

The utility of quantitative coverage statistics in ranking the sources is first explored by Florescu et. al. [FKL97]. The primary aim of both these efforts was however was on the “use” of coverage statistics, and they do 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. We do share their goal of keeping the set of statistics compact. Florescu et. al. [FKL97] achieve the compactness by assuming that each source is identified with a single primary class of queries that it exports. They “factorize” the coverage of a source with respect to an arbitrary class in terms of (a) the coverage of that source with respect to its primary class and (b) the statistics about inter-class overlap. In contrast, we consider and learn statistics about a source’s coverage with respect to any arbitrary query class. We achieve compactness by dynamically identifying “big” query classes, and keeping coverage statistics only with respect to these classes. From a learning point of view, we believe that our approach makes better sense since inter-class overlap statistics cannot be learned directly.[2]

There has also been some work on ranking text databases in the context of key word 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. First, 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.

3 Modeling Coverage and Overlap w.r.t. Query Classes

Our approach consists of grouping queries into abstract classes. In order to better illustrate the novel aspects of our association rule mining approach, we purposely limit the queries to just projection and selection queries.

3.1 Classifying Mediator Queries

Since 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

[2]They would have to be estimated in terms of statistics about the coverages of the corresponding query classes by various sources, as well as the inter-source overlap statistics. In this sense, the statistics in [FKL97] can be thought of as a post-processing factorization of the statistics learned in our framework.

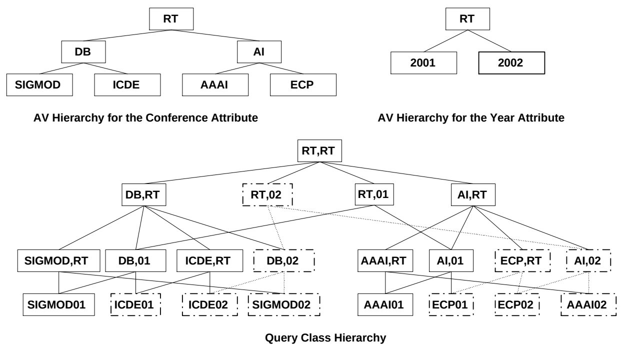


Figure 2: *AV Hierarchies and the Corresponding Query Class Hierarchy*

to the so-called "attribute value hierarchies" for a subset of the attributes of each mediated relation.

Attribute Value Hierarchies: 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(Note that, for numerical attributes, we can take value ranges as leaf nodes), while the non-leaf nodes are abstract values that correspond to the union of values below them. Figure 2 shows the AV hierarchies for the "conference" and "year" attributes of the "paper" relation. It is instructive to note that AV hierarchies can exist for both categorical and quantitative (numerical) attributes. In the case of the latter, the abstract values in the hierarchy may correspond to ranges of attribute values.

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**. If we know the domains (or representative values) of multiple attributes, we can choose as the classificatory attribute the best k attributes whose values differentiate the sources the most, where the number k is decided based on a tradeoff between prediction performance versus 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. In the latter case, we can, for instance, use decision tree learning techniques [HK00] to rank attributes in terms of their information gain in classifying the sources. For example, suppose the mediator system in Example 1 just has three sources: source S_1 only has papers in conference AAAI, S_2 only has papers in conference IJCAI, and S_3 only has papers in conference SIGMOD. In order to rank access to these sources, we need only choose the "conference" attribute as the classificatory attribute, even if we know the domain of the "year" attribute.

Once the classificatory attributes are selected, the AV hierarchies for those attributes can either be provided by the mediator designer (using existing domain ontologies, c.f.[WMY00;IGS01]), or be automatically generated through clustering techniques. In the following discussion, we will assume that AV hiearchies are made available. We discuss the issues involved in the hierarchy generation in Section 9.

Query Classes: Since we focus on selection queries, a typical 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 that are bound by the query. To classify queries that do not bind any classificatory attribute, we would have to learn simple associations [3] between the values of the non-classificatory and classificatory attributes. A query class **feature** is defined as the assignment of a specific value to a classificatory attribute 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 sharing a particular set of features. A query class having no abstract features is called a *leaf class*. The space of query classes over which we learn the coverage and overlap statistics 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 $H_1 \times H_2 \times ... \times Hn$.

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.

Example 2: Figure 2 shows an example class hierarchy for a very simple mediator with the two example AV hierarchies. The query classes are all the classes shown in the bottom half, along with the subsumption relations between the classes. For example, the class $(SIGMOD, 2001)$ refers to all the SIGMOD 2001 papers, and the class $(DB, 2001)$ refers to all the database papers

[3]A simple association would be $Author = J.Ullman \rightarrow Conference = Databases$ where $Author$ is non-classificatory while $Conference$ is a classificatory attribute

published in year 2001. As we can see the former is subsumed by the latter. Where the query class includes the root of a hierarchy (meaning that any value of that particular attribute is allowed in the query class), we suppress the root. The class $(SIGMOD, RT)$ is written as $(SIGMOD)$. □

3.2 Source Coverage and Overlap w.r.t. Query Classes

The *coverage* of a data source S with respect to a class C, denoted by $P(S|C)$, is the probability that a random tuple belonging to the class C is present in source S. We assume that the union of the contents of the available sources within the system covers 100% of the class. In other words, coverage is measured relative to the available sources. The *overlap* among a set $\widehat{S}$ of sources with respect to a class C, denoted by $P(\widehat{S}|C)$, is the probability that a random tuple belonging to the class C is present in each source $S \in \widehat{S}$.

The coverage and overlap can be conveniently computed using an association rule mining approach. Specifically, we are interested in the class-source association rules of the form $C \rightarrow \widehat{S}$, where C is a query class, and $\widehat{S}$ is a (possibly singleton) set of data sources. The overlap (or coverage when $\widehat{S}$ is a singleton) statistic $P(\widehat{S}|C)$ is simply the "confidence" of such an association rule.[4] Examples of such association rules include: $AAAI \rightarrow S_1$, $AI \rightarrow S_1$, $AI\&2001 \rightarrow S_1$ and $2001 \rightarrow S_1 \wedge S_2$.

3.3 Controlling the number of stored statistics

From the foregoing, we see that all we need to do to gather the coverage and overlap statistics is to (a) get some representative data from the sources, and categorize the data into query classes (b) mine the class source association rules from this base data. We introduce two important changes to this basic plan to control the amount of statistics learned:

Limiting statistics to "large" classes: As we discussed in Section 1, it may be prohibitively expensive to learn and store the coverage and overlap statistics for every possible query class. In

[4]Support and confidence are two measures of a rule's significance. The support of the rule $C \rightarrow \widehat{S}$ (denoted by $P(C \cap \widehat{S})$) refers to the percentage of the tuples in the global relation that are common to all the sources in set $\widehat{S}$ and belong to class C. The confidence of the rule (denoted by $P(\widehat{S}|C) = \frac{P(C\cap\widehat{S})}{P(C)}$) refers to the percentage of the tuples in the class C that are common to all the sources in $sourceSet$ $\widehat{S}$.

order to keep the number of association rules low, we would like to prune "small" classes. We use a threshold on the support of a class (i.e., percentage of the base data that falls into that class), called τ_c, to identify large classes. Coverage and overlap statistics are learned only with respect to these large classes. In this paper we present an algorithm to efficiently discover the large classes by using the *anti-monotone property*[5]([HK00]).

Limiting Coverage and Overlap Statistics: Another way we use to control the number of statistics is to only remember coverage and overlap statistics only when they are above a threshold parameter, τ_o. While the thresholds τ_c and τ_o reduce the number of stored statistics, they also introduce complications when the mediator is using the stored statistics to rank sources with respect to a query. Briefly, when a query Q belonging to a class C is posed to the mediator, and there are no statistics for C (because C was not identified as a large class), the mediator has to make do with statistics from a generalization of C that has statistics. Similarly, when a source set $\widehat{S}$ has no overlap statistics with respect to a class C, the mediator has to assume that the sources in set $\widehat{S}$ are in effect disjoint with respect to that query class. In Section 6, we describe how these assumptions are used in ranking the sources with respect to a user query. Before doing so, we first give the specifics of base data generation, discovery of large classes and coverage and overlap statistics.

In the paper, we discuss how to use the *Apriori* algorithm([AS94]) to discover strongly correlated source sets for all the large classes.

4 Gathering Base Data

In order to use association rule mining approach to learn the coverage and overlap statistics, we need to first collect a representative sample of the data stored in the sources. Since the sources in the data integration scenario are autonomous, this will involve "probing" the sources with a representative set of "probing queries." The results of the probing queries need to be organized into a form suitable for statistics mining. We discuss both these issues in this section.

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

4.1 Probing queries

We note at the outset, that the details of the rest of the steps of statistics mining do not depend on *how* the probing queries are selected. The probing queries can, however, affect the accuracy of the learned statistics in answering the queries encountered in actual practice. There are two possible ways of generating "representative" probing queries. We could either (1) pick our sample of queries from a set of "spanning queries"–i.e., queries which together cover all the tuples stored in the data sources or (2) pick the sample from the set of actual queries that are directed at the mediator over a period of time. Although the second approach is more sensitive to the actual queries that are encountered, it has a "chicken-and-egg" problem as no statistics can be learned until the mediator has processed a sufficient number of user queries.

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

Once we decide the issue of the space from which the probing queries are selected (in our case, a set of spanning queries), the next question is how to pick a representative sample of these queries. Clearly, sending all potential queries to the sources is too costly. We use sampling techniques for keeping the number of probing queries under control. Two well-known sampling techniques are applicable to our scenario: (a) *Simple Random Sampling* and (b) *Stratified Random Sampling* [M82]. Simple random sampling gives equal probability of being selected to each query in the collection of sample queries. Stratified random sampling requires that the sample population be divisible into several subgroups. Then for each subgroup a simple random sampling is done to derive the samples. We evaluate both these approaches experimentally to study the effect of sampling on our learning approach.

4.2 Efficiently managing results of probing

Once we decide on a set of sample probing queries, these queries are submitted to all the data sources. The results returned by the sources are then organized in a form suitable for mining large classes, coverage and overlap statistics. Specifically the result dataset consists of two tables, **classInfo**(CID,A_{c_1},...,A_{c_n}, Count) and **sourceInfo**(CID, Source, Count), where A_{c_j} refers to the j^{th} classificatory attribute. The leaf classes with at least one tuple in the sources are given a class identifier, CID. The total number of distinct tuples for each leaf class are entered into **classInfo**, and a separate table **sourceInfo** keeps track of which tuples come from which sources. If multiple sources have the same tuples in a leaf class then we just need to remember the total number of common tuples for that overlapped source set. An entry in the table sourceInfo for a class C and sourceset $\widehat{S}$ keeps track of the number of objects that are not reported for any superset of $\widehat{S}$. In the worst case, we have to keep the counts for all the possible subsets for each class(2^n of them, where n is the number of sources which have answers for the query)[6].

Example 3: Continuing the example in Section 1, we shall assume the following query is the first probing query:

Q(title, author, conference, year) :− **paper**(title, author, conference, year), conference=”ICDE”.

Then we can update these tuples into the dataset: **classInfo**(see Table 1) and **sourceInfo** (see Table 2). In the table **classInfo**, we use attribute CID to keep the id of the class, attributes “conference” and “year” to keep the classificatory attribute values, and attribute Count to keep the total number of distinct tuples of the class. In the table **sourceInfo**, we use attribute CID to keep the id of the class, attribute Source to keep the overlap sources in the class, and attribute Count to keep the number of overlapped tuples of the sources. For example, in the leaf class with class CID=2,

[6]Although in practice the worst case is not likely to happen, if the results are too many to remember, we can do one of the following: use a single scan mining algorithm(see Section 4.1.2), then we can count query by query during probing, in this way we just need to remember the results for the current query; just remember the counts for the higher level abstract classes; or just remember overlap counts for upto k-$sourceSet$s, where k is a predefined value($k < n$).

CID	Conference	Year	Count
1	ICDE	2002	79
2	ICDE	2001	67

Table 1: *Tuples in the table classInfo*

CID	Source	Count
1	(S_2, S_7)	79
2	(S_1, S_2, S_3)	38
2	(S_1, S_2)	20
2	S_3	9

Table 2: *Tuples in the table sourceInfo*

we have three subsets of overlapped sources which disjointly export the total 67 tuples. As we can see, all the sources in the set (S_1, S_2, S_3) export 38 tuples in common, all the sources in the set (S_1, S_2) export another 20 tuples in common, and the single source S_3 itself export another 9 tuples.

5 Algorithms for Learning Coverage and Overlap

In terms of the mining algorithms used, we already noted that the source overlap information is learned using a variant of the Apriori algorithm [AS94]. The source coverage as well as the large class identification is done simultaneously using the LCS algorithm which we developed. Although the LCS algorithm shares some commonalities with multi-level association rule mining approaches, it differs in two important ways. The multi-level association rule mining approaches typically assume that there is only one hierarchy, and mine strong associations between the items within that hierarchy. In contrast, the LCS algorithm assumes that there are multiple hierarchies, and discovers large query classes with one attribute value from each hierarchy. It also mines associations between the discovered large classes and the sources.

5.1 The LCS Algorithm

The LCS algorithm (see Figure 3) dynamically discovers the large classes inside a mediator system and compute coverage statistics for these discovered large classes. As mentioned earlier, in order to avoid too many small classes, we can set support count thresholds to prune the classes with support count below the threshold. We dynamically prune classes during counting and use the anti-monotone property to avoid generating classes which are supersets of the pruned classes. A procedure genClassSet is used to efficiently generate potentially large candidate ancestor class set for each leaf class by pruning small candidate classes using anti-monotone property.

The LCS algorithm requires the dataset: *classInfo* and *sourceInfo*, the AV hierarchies, and the minimum support as inputs. As we can see, the LCS algorithm makes multiple passes over the data. Specifically, we first find all the large classes with just one feature, then we find all the large classes with two features using the previous results and the anti-monotone property to efficiently prune classes before we start counting, and so on. We continue until we get all the large classes with all the n features. For each tuple in the k-th pass, we find the set of k feature classes it falls in, increase the count $support(C)$ for each class C in the set, and increase the count $support(r_{c \rightarrow s})$ for each source S with this tuple. We prune the classes with total support count less than the minimum support count. After identifying the large classes, we can easily compute the coverage of each source S for every large class C as follows:

$$confidence(r_{c \rightarrow s}) = \frac{support(r_{c \rightarrow s})}{support(C)}$$

In the genClassSet algorithm(see Figure 4), we find all the candidate ancestor classes with k features for a leaf class lc using procedure **genClassSet**. The procedure prunes small classes using the large class set $classSet$ found in the previous $(k-1)$ passes. In order to improve the efficiency of the algorithm, we dynamically prune small classes during the cartesian product procedure.

Example 4: Assume we have a leaf class lc={1, ICDE, 2001, 67} and k=2. We first extract the feature values $\{A_{c_1} = ICDE, A_{c_2} = 2001\}$ from the leaf class. Then for each feature, we generate a feature set which includes all the ancestors of the feature. Then we will get two feature sets: $ftSet_1 = \{ICDE, DB\}$ and $ftSet_2 = \{2001\}$. Suppose the class with the single feature "ICDE" is not a large class in the previous results, then any class with the feature "ICDE" can not

```
Algorithm LCS(classInfo; sourceInfo; $\tau_c$ : minimum support; $n$ : # of classificatory attributes)
    $classSet = \{\}, ruleSet = \{\}$;
    for($k = 1; k <= n; k++$)
        Let $classSet_k = \{\}$;
        for(each leaf class $lc \in classInfo$)
            $C_{lc} = genClassSet(k, lc, ...)$;
            for(each class $c \in C_{lc}$)
                if($c \notin classSet_k$)
                then $classSet_k = classSet_k \cup \{c\}$;
                $c.count = c.count + lc.Count$;
                for (each source $S \in lc$)
                    if (rule $r_{c \to s} \notin ruleSet$)
                    then $ruleSet = ruleSet \cup \{r_{c \to s}\}$;
                    $r_{c \to s}.count = r_{c \to s}.count+$
                        $\#$ of tuples from source S and in class $lc$;
                end for
            end for
        end for
        $classSet_k = \{c \in classSet_k | c.count >= \tau_c\}$;
        remove rules with low support classes from $ruleSet$;
        $classSet = classSet \cup classSet_k$;
    end for
    for(each rule $r_{c \to s} \in ruleSet$)
    do $r_{c \to s}.confidence = \frac{r_{c \to s}.count}{c.count}$;
    return $ruleSet$;
End LCS;
```

Figure 3: Learning Coverage Statistics algorithm

```
Procedure genClassSet($k$ : number of features;$lc$ : the leaf class; $classSet$ : discovered large class set; AV hierarchies)
    for (each feature $f_i \in lc$)
        $ftSet_i = \{f_i\};$
        $ftSet_i = ftSet_i \cup (\{ancestor(f_i)\} - \{root\});$
    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 4: Ancestor class set generation procedure

be a large 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 leaf class lc,

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

Complexity: In the LCS 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 equal to zero, we still have to keep all the classes($\prod_{i=1}^{n} |H_i|$, where H_i is the i^{th} AV hierarchy.).

However if the number of classes are small and the cost of scanning the whole dataset is very expensive, then we can use a one pass algorithm. For each leaf class lc of every probing query's results, the algorithm has to generate an complete candidate class set of lc, increase the counts of each class in the set. By doing so, we have to remember the counts for all the possible classes during the counting, but we don't need to remember all the probing query results.

5.2 Learning Overlap among Sources

Once we discover large classes in the mediator, we can learn the overlap between sources for each large class. Here we also use the dataset: **classInfo** and **sourceInfo**. In this section we discuss how to learn the overlap information between sources for a given class.

From the table *classInfo* we can classify the leaf classes into the large classes we learned using LCS. A leaf class can be mapped into multiple classes. For example, a leaf class about a publication in Conference:"AAAI", and Year:"2001", can be classified into the following classes: (AAAI,RT), (AI,RT), (RT,2001), (AAAI,2001), (AI,2001), (RT,RT), provided all these classes are determined to be large classes in the mediator by LCS.

After we classify the leaf classes in *classInfo*, for each discovered large class C, we can get its descendent leaf classes, which can be used to generate a new table $sourceInfo_c$ by selecting relative tuples for its descendent leaf classes from *sourceInfo*.

Next we apply the Apriori ([AS94]) algorithm to find strongly correlated source sets. In order to apply the Apriori on our data in $sourceInfo_c$, we do a minor change to the algorithm. Usually Apriori takes as input a list of transactions, while in our case it is a list of source sets with common tuple counts. So every time when an itemset appears in a transaction, the count of the itemset is increased by 1, while in our case, every time we find a superset of a sourceSet in $sourceInfo_c$, the count of the sourceSet is increased by the actual count of the superset.

The candidate source sets will include all the combinations of the sources, with 1-$sourceSets$, 2-$sourceSets$,...,n-$sourceSets$, where n is the total number of sources. In order to use Apriori, we have to decide a minimum support threshold, which will be used to prune uncorrelated source sets.

Once the frequent source sets from the table $sourceInfo_c$ have been found, it is straightforward to calculate the overlap statistics for these combination of strongly correlated sources. We can compute the overlap probability of these correlated sources $\{S_1, S_2, ..., S_k\}$ in class C by using the following formula:

$$P((S_1 \wedge S_2 \wedge ... \wedge S_k)|C) = \frac{support\_count(\{S_1, S_2, ..., S_k\})}{support\_count(C)}$$

Here the $support\_count(C)$ is just the total number of tuples in the table $sourceInfo_c$.

6 Using the Learned Statistics

In this section, we consider the question of how, given a user query, we can rank order the sources to be accessed, using the learned statistics.

6.1 Mapping users' queries to abstract classes

After we run the LCS algorithm, we will get a set of large classes and there is a hierarchical structure between these classes. The classes shown in Figure 2 with solid frame lines are discovered large classes. As we can see some classes may have multiple ancestor classes. For example, the class (ICDE,01) has both the class (DB,01) and class (ICDE,RT) as it's parent class. In order to use the learned coverage and overlap statistics of the large classes, we need to map a user's query to a discovered large class. Then the coverage and overlap statistics for the corresponding class can be used to predict the coverage of the sources and overlap among the sources for the query.

The mapping is done according to the following algorithm.

1. If the classificatory attributes are bound in the query, then find the lowest ancestor abstraction class with statistics[7] for the features of the query.

2. If no classificatory attribute is bound in the query, then we do one of the following,

    - Check whether we have learned some association rules between the non-classificatory features in the query with classificatory features[8]. If we did, we use these features as features of the query to get statistics, go to step 1;

[7] If we have multiple ancestor classes, the lowest ancestor class with statistics means the ancestor class with lowest support counts among all the discovered large classes.

[8] In order to simplify the problem, we did not discuss this kind of association rule mining in this paper, but it is just a typical association rule mining problem. A simple example would be to learn the rules like: $J.Ullman \rightarrow Databases$ with high enough confidence and support.

- Present the discovered classes to the user, and take the user's feedback to select a class;
- Use the root of the hierarchy as the class of the query.

6.2 Computing residual coverage

In this section we discuss how we compute the residual coverages in order to rank the sources for the class C (to which the user's query has been mapped), using the learned statistics. In order to find a plan with top k sources, we start by selecting the source with the highest coverage ([FKL97]) as the first source. We then we use the overlap statistics to compute the residual coverages of the rest of the sources to find the second best, given the first; the third best, given the first and second, and so on, until we get a plan with the desired coverage.

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

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

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

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

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

A naive evaluation of this formula would require 2^n accesses to the database of learned statistics, corresponding to the overlap of each possible subset of the n sources with source S. It is however possible to make this computation more efficient by exploiting the structure of the stored statistics. Specifically, recall that we only keep overlap statistics for 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 andS_4 are sources exporting tuples for class C. Let $P(S_1|C)$, $P(S_2|C)$ $P(S_3|C)$ and $P(S_4|C)$ be the learned coverage statistics, and $P(S_1 \wedge S_2|C)$

and $P(S_2 \wedge S_3|C)$ be the learned overlap statistics. The expression for computing the residual coverage of S_3 given that S_1 and S_2 are already selected is:

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

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

The inputs to the algorithm in Figure 5 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 Setup

We have implemented the statistics learning system *StatMiner* as part of *Havasu* [KNNV02], a prototype system supporting query processing for Web data integration. Havasu system is designed to support multi-objective query optimization [NK01], flexible execution and mining strategies for learning autonomous Web source statistics. Given a global mediator schema and Web sources exporting the schema, our approach for learning source statistics aims at capturing an approximate distribution of the source data that would enable efficient processing of potential mediator queries. We use the AV hierarchies provided for the global schema to generate a hierarchical classification of potential queries and also to generate sample probing queries to learn the approximate data

```
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 $\widehat{S}_c \cup \{\mathrm{s}\}$
   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$(\mathrm{s}, \widehat{S}_s, \widehat{S}'_c)$;
    end for
    $resCoverage = resCoverage + (-1)^{|\widehat{S}_c|} overlap$;
  end if
  return $resCoverage$;
End residualCoverage;
```

Figure 5: Algorithm for computing residual coverage

spread of the sources with respect to the classification. To evaluate our techniques we set up a set of “remote” data sources accessible on the Internet. The sources were populated with two types of data. The *TPC* sources were populated with synthetic data generated using the data generator from TPC-W benchmark [TPC] (see below). The TPC sources support controlled experimentation as their data distribution (and consequently the coverage and overlap among web sources) can be varied by us. The second type of sources, called *DBLP* sources, were populated with data from the DBLP repository [DBLP].

7.1 Data Sources

TPC Sources: We designed 25 sources using 200000 tuples for the relation Books. We chose **Books**(Bookid, Pubyear, Subject, Publisher, Cover) as the relation exported by our sources. The decision to use Books as the sample schema was motivated by the fact that multiple autonomous Internet sources projecting this relation exist, and in the absence of statistics about these sources, only naive mediation services are currently provided. Pubyear, Subject and Cover are used as the *classificatory* attributes in the relation Books. The hierarchies were designed as shown in Figures 6 and 7. To evaluate the effect of the resolution of the hierarchy on ranking accuracy we designed two separate hierarchies for Subject, containing 180 and 40 leaf nodes respectively. Leaf node values for Pubyear range from 1980 to 2001, while Cover is relatively small with only five leaf nodes. The Subject hierarchy was modeled from the classification of books given by the online bookstore Amazon [AM]. We populated the data sources exporting the mediator relation using DataGen, the data generator from TPC-W Benchmark [TPC]. The distribution of data in these sources was determined by controlling the values used to instantiate the classificatory attributes Pubyear, Subject and Cover. For example, two sources S_1 and S_2 both providing tuples under abstract feature “Databases” of Subject hierarchy, are designed to have varying overlap with source S_3, by selecting different subsets of features under “Databases” to instantiate the source tuples. These subsets may be mutually exclusive, but they overlap with the subset of features selected for populating source S_3. Since the actual generation of data sources is done by using DataGen, the above mentioned procedure gives us a macro level control over the design of overlap among sources. In fact DataGen populates the sources by initializing each attribute of the Books relation using a randomly chosen value from a list of seed values for that attribute. Hence we control the query classes for which the sources provide answer tuples and may overlap with other sources but not the actual values of coverage and overlap given by sources.

DBLP Sources: We generated 15 medium-sized data sources as materialized views over *DBLP*. The views were designed to focus on publications in Artificial Intelligence and Database research only. These sources export the mediator relation **paper**(title, author, conference, year). We designed simple hierarchies for “conference” and “year” attribute. We use the DBLP sources to evaluate our ability to learn an approximate data distribution from real Web data and also to test the effect of various probing techniques we use.

7.2 Probing Data Sources

To construct the data spread of autonomous Web sources we must probe these sources. As discussed in Section 4.1 we use the leaf values of the AV hierarchies to generate a set of spanning mediator queries. Specifically we generate the sample queries by taking cartesian products of the leaf node values. Although we are able to generate sample queries using the AV hierarchies, the number of queries can become quite large for large sized or large number of hierarchies.

Query Sampling: As mentioned in Section 4.1, we generate the set of sample probing queries using both *Simple Random Sampling* and *Stratified Random Sampling* [M82]. After generating the set of spanning queries we use the two sampling approaches to extract a sample set of queries to probe the data sources. Simple Random sampling picks the samples from the complete set of queries, whereas to employ the Stratified Random sampling approach, we have to further classify the queries into various *strata*. The strata is chosen as the abstract feature of any one classificatory attribute say $A1$. All the queries that bind $A1$ using leaf values subsumed by a strata are mapped to that strata. A strata based on an abstract feature that only subsumes leaf nodes will have fewer queries mapped to it compared to the strata that is based on an abstract feature that subsumes both the leaf nodes and other abstract features. Thus the level of abstraction at which we decide a strata varies the number of queries that get mapped to the strata. The lowest abstraction is the leaf node, while the root gives highest abstraction. Selecting root as the strata will make Stratified Random Sampling equal to Simple Random, where selecting the leaf nodes as strata, will be equal to issuing all the spanning queries. The cost of probing is directly proportional the number of probing queries issued. Hence choosing a good strata that requires low probing while achieving a good approximation of the data spread of sources becomes a challenge.

7.3 Learning Efficiency

To evaluate the accuracy of the statistics learnt by *StatMiner* we tested them using two simple plan generation algorithms implemented as part of *Havasu*. Our mediator implements the **Simple Greedy** and **Greedy Select** algorithms described in [FKL97] to generate query plans using the

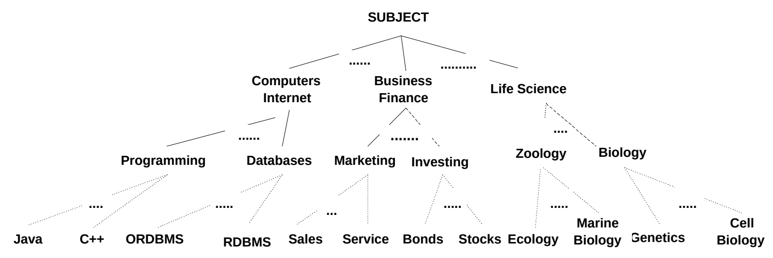


Figure 6: Subject Hierarchy

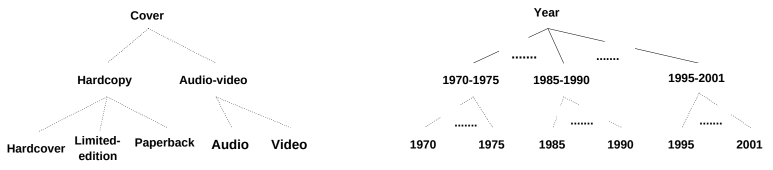


Figure 7: Cover and Year Hierarchy

source coverage and overlap statistics learnt by *StatMiner*. Given a query, Simple Greedy generates a plan by assuming all sources are independent and greedily selects top k sources ranked according to their coverages. On the other hand, Greedy Select generates query plans by selecting sources with high residual coverages calculated using both the coverage and overlap statistics (see Section 6.2). We evaluate the plans generated by both the planners for various sets of statistics learnt by *StatMiner* for differing threshold values and AV hierarchies. We compare the precision of plans generated by both the algorithms. We define the **precision** of a plan as a fraction of number of sources in the plan which turn out to be true top k sources that can give the highest cumulative coverage for the query. We determine the true top k sources by querying all the sources exporting the mediator relation and ranking them in terms of the unique tuples they provide.

8 Experimental Results

In this section we present results of experiments conducted to study the variation in pruning power and accuracy of our algorithms for different class size thresholds τ_c. In particular, given a set of sources and probing queries, our aim is to show that we can trade time and space for accuracy by increasing the threshold τ_c. Specifically by increasing threshold τ_c, the ***time*** (to identify large classes) and ***space*** (number of large classes remembered) usage can be reduced with a reduction in ***accuracy*** of the learnt estimates. All the experiments presented here were conducted on a 500MHZ

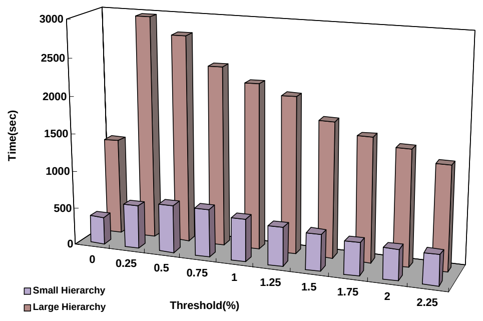


Figure 8: *LCS learning time for various thresholds*

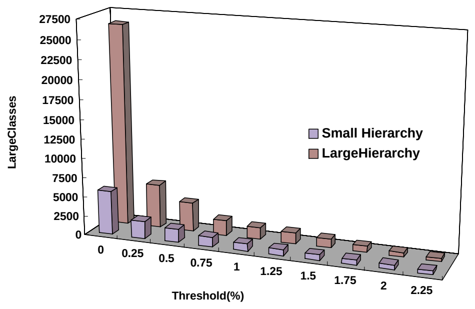


Figure 9: *Pruning of classes by LCS*

Sun-Blade-100 systems with 256MB main memory running under Solaris 5.8. The sources in the mediator are hosted on a Sun Ultra 5 Web server located on campus.

8.1 Results over TPC Sources

Effect of Hierarchies on Space and Time: To evaluate the performance of our statistics learner, we varied τ_c and measured the number of large classes and the time utilized for learning source coverage statistics for these large classes. Figure 8 compares the time taken by LCS to learn rules for different values of τ_c. Figure 9 compares the number of pruned classes with increase in value of τ_c. We represent τ_c as a percentage of the total number of tuples in the relation. The total tuples in the relation is calculated as the number of unique tuples generated by the probing queries.

As can be seen from Figure 8, for lower values of threshold τ_c, LCS takes more time to learn the rules. For lower values of τ_c, LCS will prune less number of classes and hence for each class in ClassInfo, LCS will generate large number of rules. This in turn explains the increase in learning time for lower threshold values.

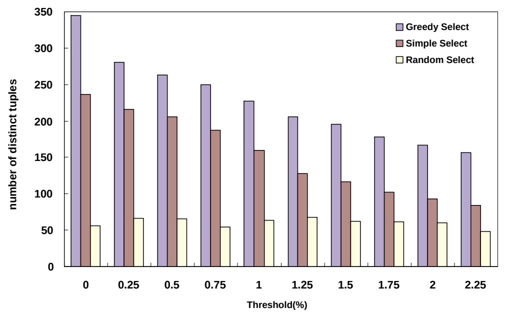


Figure 10: *Comparing Plan Coverages*

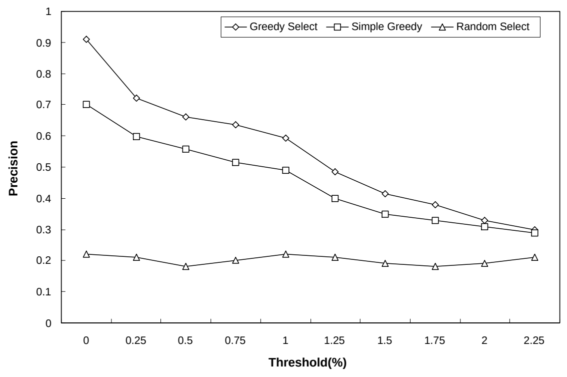


Figure 11: *Comparison of Plans Using Learned Estimates*

In Figure 9, with increase in value of τ_c, the number of small classes pruned increase and hence we see a reduction in the number of large classes. For any value of τ_c greater than the support of the largest abstract class in the classSet, LCS returns only the root as the class to remember. Figures 8 and 9 show LCS performing uniformly for both Small and Large hierarchy. For both hierarchies, LCS generates large number of classes for small threshold values and requires more learning time. From Figures 8 and 9, we can see that the amount of time used and classes generated (space requirement) for the Large hierarchy is considerably higher than for Small hierarchy.

Accuracy of Estimated Coverages: To calculate the error in our coverage estimates, we used the prototype implementations of "Simple Greedy" and "Greedy Select" algorithms under *Havasu* Web data integration system and a subset of our probing queries as test queries. Since the test queries will have classificatory attributes bound,from Section 6.1 we see that the integration engine maps it to the lowest abstract class for which coverage statistics have been learnt. Once the query is mapped to a class, the mediator then generates plans using the ranking algorithms, Simple Greedy and Greedy Select as described in Section 7.3. We compare the plans generated by these algorithms with a naive plan generated by **Random Select**. Random select algorithm arbitrarily picks k sources without using any statistics. The source rankings generated by all the three algorithms is compared with the "true ranking" determined by querying all the sources. Figure 11 compares the precision of plans generated by the three approaches with respect to the true ranking of the sources.

Once a plan is chosen, the query engine then issues the query to sources in descending order of their rank in the plan. Suppose the testing query is $Subject = ORDBMS$, and the statistics are available for class $Databases$ while class $ORDBMS$ was pruned by LCS. From Figure 6 we can see that $Databases$ is the next lowest abstract class for the query and hence the coverage statistics for $Databases$ would be used to generate a plan.

As can be seen from Figure 10 for all values of τ_c Greedy Select gives the best plan, while Simple Greedy is close second, but the Random Select performs poorly. The results are according to our expectations, since Greedy Select generates plans by calculating residual coverage of sources and thereby takes into account the amount of overlap among sources, while Simple Greedy calls sources with high coverages thereby ignoring the overlap statistics and hence generates less number of tuples.

In Figure 11 we compare the precision of plans generated by the three approaches. We define the precision of a plan to be the fraction of sources in the estimated plan, which turn out to be the real top k sources after we execute the query. Figure 11 shows the precision for the top 5 sources in a plan. Again we can see that Greedy Select comes out the winner. The decrease in precision of plans generated for higher values of threshold can be explained from Figure 9. As can be seen, for larger values of threshold more number of leaf classes get pruned. A mediator query always maps to a particular leaf class. But for higher thresholds, the leaf classes are pruned and hence queries get mapped to higher level abstract classes. Therefore the statistics used to generate plans have lower accuracy and in turn generates plans with lower precision.

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

8.2 Results over DBLP Sources

Effect of Probing Strategies: Given that the cost of probing tends to dominate the statistics gathering approach, we wanted to see how accurate the learned statistics are with respect to the two probing strategies. We used *DBLP sources* projecting the relation *paper*(title, author, conference, year) for evaluating the probing strategies. We generated AV hierarchies for attributes "conference" and "year". The set of probing queries are generated by taking a cartesian product of the values of these AV hierarchies. The total number of queries generated is 225 and we call them the set of "All queries" in our experiments. Initially we queried all the sources using "All queries". We then learn the coverage and overlap statistics for all the sources using LCS for different values of thresholds τ_c. Once the values of coverage and overlap are determined we use them as a baseline to evaluate the performance of our approach with respect to the different probing methods. Figure 12 compares the effect of different probing strategies on LCS in terms of its ability to prune large classes. We generated probing queries containing either 48 or 96 queries using both the sampling methods. For the experiments reported here we decided to use a limit on the total number of queries selected for probing as a stopping criteria for sampling. Better approaches based on cost of probing are currently being pursued.

As seen from Figure 12, for the probing queries under "Stratified 96", LCS shows the closest performance to that of "All queries". Since stratified sampling selects the queries from the strata or subclasses, it can ensure better representation for the classes in the AV hierarchies in terms of the queries that are mapped to them. In doing so it is able to get a better approximation of the data spread of the sources. In comparison, random sampling selects the queries from the entire set of queries and therefore can be biased in terms of the classes represented by the selected queries. Figure 12 also shows that the efficiency of learning can be improved given larger sized probing queries which is intuitive. "Stratified 96" gives better pruning performance compared to "Stratified 48" but requires double the number of probing queries. The cost of probing is less for smaller set of probing queries. Hence the improved pruning shown by LCS for "Stratified 96" entails a higher cost of probing compared to "Stratified 48".Hence we must achieve a delicate balance between

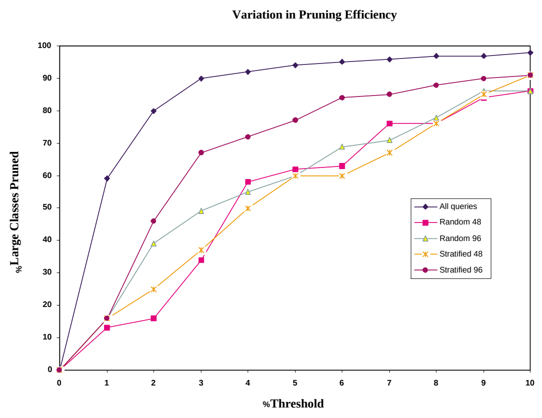


Figure 12: *Variation in Pruning Efficiency with Probing*

the number of probing queries issued versus how closely we approximate the data spread of the sources.

In Figure 13, we compare the accuracy of our approach in learning large classes based on the results of probing queries generated by the both the Sampling strategies. Here we compared the actual large classes learnt by LCS based on the probed results of the various sampling strategies with that generated by issuing all the queries to the sources. We provide the number of large classes to be learnt as the threshold to LCS. Figure 13 shows the difference in terms of the actual large classes identified by LCS for the various sets of queries. As can be seen for queries in the set "Stratified 96" LCS learns best i.e, it almost learns the same set of large classes as learnt for "All queries". When the number of large classes learnt remembered is 30, we get a mismatch of less than 5 classes using our learning approach.

The above results are encouraging and show our approach is able to give good results even for a smaller sample of probing queries. We are looking at better sampling strategies to use and also at identifying better heuristics to decide the stopping criteria for the sampling criteria.

9 Discussion and Future Directions

In this section, we will discuss some practical issues regarding the realization of the *StatMiner* approach, and outline several future directions for this work.

Generating AV Hierarchies: Classification hierarchies similar to AV hierarchies are assumed to be available in other research efforts such as [WMY00,IGS01]. Nevertheless, generating the AV

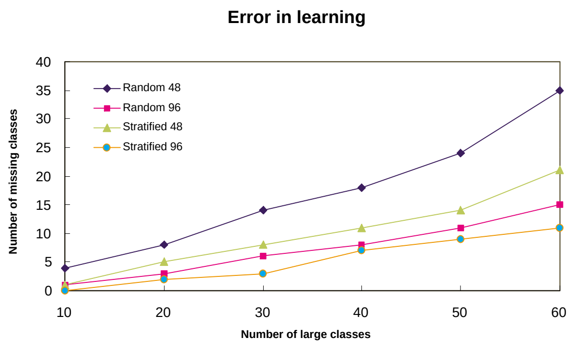


Figure 13: *Variation in large class estimation with Pruning*

hierarchies is a semi-manual activity, and can thus present practicial difficulties. In the following, we discuss some factors that mitigate these difficulties. To begin with, hierarchy design will be easy in cases where the attribute values correspond to an existing ontology([WMY00],[IGS01]). Secondly, AV hierarchies are easy to build for quantitative attributes such as the Year attribute in our example relation. Thirdly, we can use some existing clustering algorithms([HK00]) to automatically generate AV hierarchies(especially for numerical attributes).[9] Finally, we may not need completely accurate AV hierarchies. We are currently extending our approach to support adaptively modifying a poor AV hierarchy as we get more user queries. The frequently asked queries that are not in the AV hierarchies will be reported and included into the corresponding hierarchies.

Choosing the *StatMiner* threshold parameters: As discussed earlier, the *StatMiner* algorithms take two threshold parameters: τ_c and τ_o. Both the parameters can be set by the administrator to strike an appropriate balance between the amount of statistics remembered and the accuracy of the statistics. In the following, we summarize some guidelines for setting the parameters. The threshold τ_c is used to decide whether a query class has large enough support to be remembered. If τ_c is set to be 0, then every possible query class will be remembered. In this way we can get the most accurate coverage statistics. If τ_c is set to be 1, only the root class will be remembered. In this way all the queries will be mapped to the root class, and the coverage estimation of some queries may be very inaccurate. The administrator can select a number between 0 and 1 as τ_c according to the space available for remembering the coverage statistics and the performance tradeoffs between

[9]It is of course possible to further automate the AV hierarchy generation–using agglomerative clustering techniques [HK00], and we are in fact investigating such an alternative. One disadvantage of such a method however is that the generated hierarchies may not have any meaning to the user of the mediator.

the additional time need for searching the statistics for a query and the time gained by using more accurate statistics to answer the query.

Another threshold τ_o is used to decide whether or not the overlap statistics between a set of sources and a remembered query class should be stored. If τ_o is set to be 0, then for each discovered large class, the overlap statistics for every source set will be remembered. If τ_o is set to be 1, then for a discovered large class, at most one source set(if all the souces are completely overlaped for the class) will be remembered. The above discussion for seting τ_c can also be applied to set the parameter τ_o.

Statistics for Handling Join Queries: In this paper, we focussed on learning learn coverage and overlap statistics of select and project queries. 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. Specifically, the following issues may have to be re-considered to support join queries:

1. Classificatory attributes selection: Instead of selecting classificatory attributes from a single relation, we will need to select attributes among all the relations in the join query;

2. Probing the sources: We can use the leaf nodes of a AV hierarchy to probe the sources of the first relation, and use the results of the first relation to probe the sources of the second relation, and so on. For each relation of the query, we use a classInfo and sourceInfo table to remember the counts of the relation. We count only the tuples that are the join results of the query. We can consider the join results as one big relation, and join query can be considered as the select and project query of this big relation.

3. Discovering large join query classes: Once we have the classificatory attributes and the join result relation, we can use the LCS algorithm to discover large classes.

4. Computing the coverage statistics for each relation: For each relation in the join query, we can compute the coverage and overlap statistics using the corresponding result relation.

Combining Coverage and Response-time Statistics: In the current paper, we assumed a simple coverage-based cost model to rank the available Websources for a query. However users may be interested in plans that are optimal w.r.t. any of a variety of possible combinations of different objectives. For example, some users may be interested in fast execution with reasonable coverage, while others may require high coverage even if with higher execution cost. Users may also be interested in plans that produce answer tuples at a steady clip (to support pipelined processing) rather than all at once at the end. In [NK01], we present the *Multi-R* query planning framework, that uses the gathered coverage and response time statistics to support multi-objective query optimization in data integration. The query planning techniques used in *Multi-R* are able to output query plans satisfying a variety of coverage and response-time tradeoffs, and still manage to keep "planning time" (i.e. search for query plans) within reasonable limits, despite the increased complexity of optimization. Our ongoing work on the *Havasu* prototype data integration system combines the *Multi-R* query planning framework and the *StatMiner* statistics learning approach to provide a comprehensive query processing methodology in the presence of Websources.

10 Conclusion

In this paper we motivated the need for automatically learning the coverage and overlap statistics of sources for efficient query processing in a data integration scenario. We then presented a set of connected techniques that estimate the coverage and overlap statistics while keeping the needed statistics tightly under control. Our specific contributions include:

- A model for supporting a hierarchical classification of the set of queries.
- An approach for estimating the coverage and overlap statistics using association rule mining techniques.
- A threshold-based modification of the mining techniques for dynamically controlling the resolution of the learned statistics.

We described the details and implementation of our approach. We also presented an empirical evaluation of the effectiveness of our approach in a realistic setting. Our experiments demonstrate that:

- We can systematically trade time and space consumption of the statistics computation for accuracy by varying the large class thresholds.

- The learned statistics provide tangible improvements in the source ranking, and the improvement is proportional to the type (coverage alone vs. coverage and accuracy) 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.

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

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

[CGHI94] S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantino, J.Ullman, J. Widom. The TSIMMIS project: Integration of heterogeneous information sources. In *IPSJ*, Japan, 1994.

[DL99] A. Doan and A. Levy. Efficiently Ordering Plans for Data Integration. The *IJCAI-99 Workshop on Intelligent Information Integration*, Stockholm, Sweden, 1999.

[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)

[KNNV02] S. Kambhampati, U. Nambiar, Z. Nie and S. Vaddi. Havasu: A multi-objective, adaptive query processing framework for data integration. ASU CSE TR-02-005 http://rakaposhi.eas.asu.edu/havasu.html

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

[IGS01] P. Ipeirotis, L. Gravano, M. Sahami. Probe, Count, and Classify: Categorizing Hidden Web Dababases. 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.

[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.

[NKNV01] Z. Nie, S. Kambhampati, U. Nambiar and S. Vaddi. Mining Source Coverage Statistics for Data Integration. Proc. WIDM(CIKM workshop) 2001.

[NNVK02] Z. Nie, U. Nambiar, S. Vaddi and S. Kambhampati. Mining Coverage Statistics for Websource Selection in a Mediator. To appear in ACM 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.

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

[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.