

# Building XML Statistics for the Hidden Web

Ashraf Aboulnaga  
IBM Almaden Research Center  
aashraf@almaden.ibm.com

Jeffrey F. Naughton  
University of Wisconsin - Madison  
naughton@cs.wisc.edu

## ABSTRACT

There have been several techniques proposed for building statistics for static XML data. However, very little work has been done in the area of building XML statistics for data sources that export XML views of data that is stored in relational or other databases. For such data sources, we need statistics that are built in an *on-line* manner, by observing the XML queries to the data sources and their results. In this paper, we present a technique for building on-line XML statistics by observing the XPath queries issued to a data source and their result sizes. These XPath queries select parts of the *virtual XML document* representing the XML view of the data at the data source. We convert these XPath queries to a more abstract and generalized form that we call *annotated path expressions*. We present a technique for storing these annotated path expressions and information about their selectivity for use in estimating the selectivity of future XPath queries. We also present an experimental evaluation of our proposed approach.

## Categories and Subject Descriptors

H.2 [Database Management]: Systems

## General Terms

Algorithms, Performance, Design

## Keywords

query optimization, selectivity estimation, database statistics, XML, hidden web

## 1. INTRODUCTION

There is currently a lot of interest in developing Internet query processors that can “query the Web.” Since XML is becoming the standard data representation format for Web data, these query processors can be built assuming that all the data that they query will be in XML format. Examples of systems that query XML data over the Internet include Niagara [13] and Xyleme [20].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

CIKM'03, November 3–8, 2003, New Orleans, Louisiana, USA.

Copyright 2003 ACM 1-58113-723-0/03/0011 ...\$5.00.

These Internet query processors can easily query data that is in XML files on the Web. We call this *static XML data*. However, most of the data on the Web is *not* in static XML files, or even HTML files. Most of the data on the Web is “hidden” in databases and can only be accessed by posing queries over these databases [6, 16]. This portion of the Web is known as the *hidden Web*. Sometimes it is also referred to as the *deep Web*. We can expect that, in the near future, hidden Web data sources will export the data they produce in response to user queries in XML format. It should therefore be possible for Internet query processors like Niagara to query the hidden Web in addition to the “static Web.”

Querying the hidden Web is of particular importance because the size of the hidden Web is up to 400 to 500 times larger than the size of the static Web. Furthermore, data in the hidden Web is typically very high-quality data [6]. Examples of hidden Web data sources include the FactFinder database of census information from the U.S. Census Bureau and the EDGAR database of company financial statements from the Securities and Exchange Commission.

Querying the hidden Web requires the ability to optimize queries over hidden Web data sources, which requires *XML statistics* about these sources. Existing solutions fall short of providing such statistics, and in this paper we propose a solution to this problem.

As a motivating example, consider the query in Figure 1, expressed in XQuery [5]. This is a join query that asks for price quotes under \$25,000 from car dealers in Madison for year 2003 cars that received a 5 star rating in the government crash tests. The URLs in this query are for actual Web sites that can be queried to provide the required information. The interface for queries and responses at these sites is currently HTML not XML. However, it would be reasonable to expect that this interface may become XML in the near future, thereby making such a query feasible. Furthermore, it is always possible to build *wrappers* around Web sites to provide an XML view of their HTML data.

The query in Figure 1 uses XPath [4] path expressions to query the hidden Web data sources. In this paper, we focus on building statistics that can help in estimating the selectivity of such XPath path expressions. The typical approach for building statistics for relational or XML data is to scan the entire data and summarize it in some data structure that occupies a small amount of memory. This does not work for the hidden Web because *we do not have access to the data*. We only have access to *queries* on the XML view of this data and their results. Most hidden Web data is stored in relational databases, and sometimes in document databases. It

```

FOR $r IN document("http://www.nhtsa.gov/")//safety/car[year=2003 and rating=5]
  $q IN document("http://autos.yahoo.com/")//newcar/quote[city="Madison"]
WHERE $r/make=$q/make and $r/model=$q/model and $q/price<25000
RETURN $q/dealer

```

Figure 1: An example query in the XQuery language

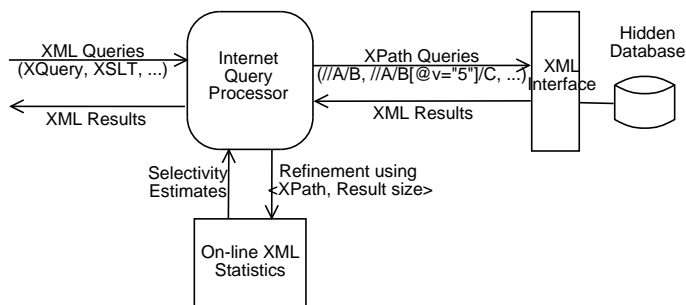


Figure 2: On-line XML statistics

is *not* stored as native XML. Rather, the XML view of the data is computed only in response to user queries. Thus, we cannot scan the entire data to build statistics. Moreover, even if this data were to be fully converted to native XML (which is highly unlikely), we would still not have access to the entire data due to proprietary rights. The owners of data are typically not willing to export their entire data, even if they are willing to export answers to queries over this data. For example, we can easily get the price of an individual book from Amazon.com but not their entire price list.

For this environment, we need *on-line XML statistics*, which are constructed by *observing user queries to hidden Web data sources and their results*. In this paper, we propose a type of on-line XML statistics which we call *on-line annotated path tables*. We do not require any cooperation from the data sources in building these statistics; the statistics are based solely on feedback from user queries. To reduce the overhead and complexity of the statistics, we use only the *sizes* of the query results (i.e., the number of XML elements they contain) for constructing the on-line XML statistics, and not the actual XML data in these results.

Thus, the problem we are trying to solve is as follows: Given a sequence of XPath queries on a *virtual XML document* representing the XML view of the data at a hidden Web data source, and given the result sizes of these queries, construct on-line XML statistics for this data source (Figure 2). The statistics should leverage the information obtained from past XPath queries to estimate the selectivity of future XPath queries issued to the data source, including XPath queries that are seen for the first time. Selectivity estimation accuracy should increase as more queries are observed. Also, there should be a mechanism for bounding the amount of memory consumed by the statistics to any given value. We consider XPath path expressions of the form  $//a_1/a_2/\dots/a_n$ . Each step,  $a_i$ , of a query path expression is either of the form  $t_i$ , where  $t_i$  is a tag name, or of the form  $t_i[c_i]$  where  $t_i$  is a tag name and  $c_i$  is an arbitrarily complex condition. Examples of such XPath queries include  $//safety/car[make="Saturn" and year=2003]/rating$ , and  $//chapter[@title="Introduction"]/section[1]$ .

The rest of this paper is organized as follows. Section 2 presents an overview of related work. Section 3 introduces

*path annotations*, which we use in our on-line statistics. Section 4 describes these statistics. Section 5 presents an experimental evaluation of our proposed technique. Section 6 contains concluding remarks.

## 2. RELATED WORK

Many different techniques for building statistics for *static* XML data have recently been proposed. The techniques in [7] build statistics that are used to estimate the selectivity of *twig queries*, or branching path expressions. The techniques we propose in [1] provide more accurate selectivity estimates for the case of simple path expressions, which are path expressions that have one branch and navigate in the XML data based on structure, without conditions. [9] proposes a statistics framework that leverages information from the XML schema of a document. [14] proposes a synopsis data structure for *graph structured* XML data, and [15] extends this framework to handle values. While these papers represent significant advances in the area of statistics for static XML data, the techniques they propose are not applicable to the hidden Web as they require access to the data, and sometimes its schema, to construct the statistics.

To our knowledge, the only paper proposing an on-line technique for constructing XML statistics is [11]. The technique proposed in that paper can only handle very simple conditions in the path expressions of the form  $\text{tag}=\text{value}$ , whereas we focus on handling complex conditions. Also, the technique in [11] only captures local information about the steps of the path expressions, whereas our technique captures information about entire path expressions.

Querying multiple hidden Web data sources in an Internet query processor is similar to querying multiple data sources in *data integration systems* such as Tukwila [10], Garlic [17], or HERMES [3]. Data integration systems optimize and execute queries over diverse data sources, so they must address the problem of obtaining statistics for these sources.

Some systems require the data sources to explicitly export the statistics required for query optimization [12, 17]. This is not applicable to our problem of building statistics for the hidden Web, because the hidden Web data sources are autonomous and provide no information beyond answers to user queries. Moreover, the data sources themselves may not have the required information about the XML data, because this data is materialized only in response to user queries.

Another approach is to design the data integration system to allow for run-time re-optimization of queries [10]. This approach assumes that the query optimizer will have little or no statistics about the data sources, so it may choose an inefficient query execution plan. As the plan is executed, more information about the data sources is obtained, and the query processor may choose to re-optimize the query based on this new information. Providing statistics at query optimization time, as we do in this paper, helps the query optimizer choose a good plan from the outset. It may still be possible to improve the performance of the query by

run-time re-optimization, although the need for such re-optimization will be less because the initial plan is good.

The HERMES system records the result sizes of queries issued to data sources and uses the recorded values to estimate the selectivity of future queries issued to these sources [3]. We also use the result sizes of queries to build statistics, but we focus on XML path expressions over hidden Web data sources, while the HERMES system focused on function calls to external programs or data sources in a distributed mediator system. Their techniques for gathering, summarizing, and using statistics do not extend to our problem.

### 3. PATH ANNOTATIONS

A simple way of building on-line XML statistics would be to cache the XPath queries issued to a data source and their result sizes. This way, if we see an XPath query for the second time, we could find its exact selectivity from the query cache. However, this technique must cache *every* query and its result size, so it does not scale in the number of queries. Furthermore, this technique is of no use for XPath queries that are seen for the first time.

Instead, our approach is to *syntactically analyze* the XPath path expressions and convert them into a more abstract and general representation that we call *annotated path expressions*. We then store these annotated path expressions and information about their result sizes for use in selectivity estimation. An annotated path expression represents *all* XPath path expressions that have a particular *form*, so it provides a degree of summarization and generalization to previously unseen XPath queries. The intuition behind annotated path expressions is that it is unlikely that we will see the exact same XPath query over and over in a query workload, but it is highly likely that we will see XPath queries of the same form repeated in the workload. Next, we describe the two types of path annotations used in our approach: *condition annotations* and *structure annotations*.

#### 3.1 Condition Annotations

An important issue we must address is how to deal with *conditions* in the XPath path expressions in our on-line XML statistics. On the one hand, we must allow conditions in the XPath queries that we consider. Without conditions, users would be able to express only a very limited and weak form of queries. For example, without conditions, users would be able to ask a car safety data source for “the safety rating of all cars” (`//safety/car/rating`) but not for “the safety rating of 2003 Saturns” (`//safety/car[make="Saturn" and year=2003]/rating`).

On the other hand, conditions complicate the construction of statistics because we cannot ignore their effect, nor can we isolate it. The selectivity of the query `//safety/car[make="Saturn" and year=2003]/rating` is much smaller than the selectivity of the unconditional query `//safety/car/rating`. Thus, we cannot ignore the effect of the condition `[make="Saturn" and year=2003]`. But at the same time, the effect of such a condition on selectivity cannot be isolated, since we only have the number of `//safety/car/rating` elements that satisfy the condition `[make="Saturn" and year=2003]`, but not the total number of `//safety/car/rating` elements.

To solve this problem, we analyze the *syntax* of the conditions in the different steps of the XPath path expressions, and we classify these conditions based on their *syntactic*

*form*. For example, we can classify conditions into either “conditions that are based only on the structure of the XML data,” such as the condition in `car[rating]` (a car that has a safety rating), or “conditions that are based on the element or attribute values in addition to the structure of the data,” such as the condition in `car[rating=5]` (a car with a safety rating that has the value 5). These are two different *forms* of conditions in the XPath steps.

We use this syntactic classification to deal with conditions by making the assumption that *conditions of a particular form have a uniform effect on selectivity*. This means that a condition of a particular form on a tag in an XPath path expression has the same effect on selectivity as any other condition of the same form on this tag. We reflect this assumption by *annotating* every tag in a path expression with an annotation describing the form of the condition on this tag. Thus, if we classify conditions as above into “conditions on structure” and “conditions on values,” we would annotate a tag,  $A$ , as follows: If  $A$  has no condition, we would annotate it with  $U$ , for *unconditional*. If  $A$  has a condition on *structure*, such as  $A[B]$ , we would annotate it with  $C_S$ . If  $A$  has a condition involving element or attribute *values*, such as  $A[B = 5]$ , we would annotate it with  $C_V$ . Thus, for the purpose of selectivity estimation, the tag,  $A$ , is represented as  $A^U$ ,  $A^{C_S}$ , or  $A^{C_V}$ . We treat  $A^U$ ,  $A^{C_S}$ , and  $A^{C_V}$  as *three distinct tags*, thereby reflecting our assumption that all conditions of a particular form have the same effect on selectivity but that conditions of different forms have different effects on selectivity.

This framework of syntactically classifying conditions based on their form and annotating the steps of the path expressions to reflect this classification allows for a wide range of classification and condition annotation schemes. The goal of any scheme should be to group conditions into classes based on syntactic analysis, such that all conditions in the same class have the same effect on selectivity, and conditions in different classes have different effects on selectivity.

An alternative to the classification scheme outlined above is to classify conditions into three classes: “simple conditions” with no logical operators, such as `[B=5]`; “conjunctive conditions” that involve only the logical operator “and,” such as `[B=5 and C=7]`; and “disjunctive conditions” that have one or more instances of the logical operator “or,” such as `[B=5 or C=7]`. It is intuitively clear that these different classes of conditions can have different effects on selectivity. For this classification scheme, we would annotate tags in the XPath path expressions with  $U$ ,  $C_S$ ,  $C_A$ , or  $C_O$  for unconditional, simple condition, conjunctive condition with only “and,” or disjunctive condition with some “or,” respectively.

Another alternative would be to opt for maximum simplicity and classify tags into “tags without a condition,” which would be annotated with a  $U$ , and “tags with a condition,” which would be annotated with a  $C$  (i.e., group all conditions into one class). This classification scheme allows for the maximum possible generalization from observed XPath queries to future XPath queries. When we observe an XPath query with some condition on a certain tag, we can use the selectivity information obtained from this query to estimate the selectivity of other similar XPath queries that have *any condition* on this tag.

In any condition classification and annotation scheme, having more classes increases the granularity with which we partition the space of possible conditions, but it de-

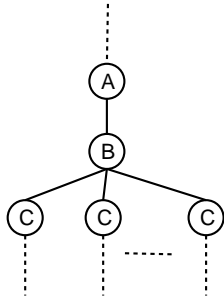


Figure 3: Minimum number of ancestors for  $C$

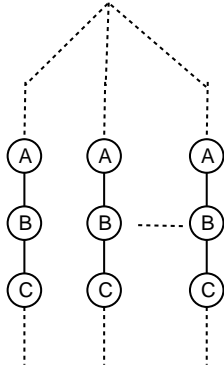


Figure 4: Maximum number of ancestors for  $C$

increases the degree of generalization from conditions in observed XPath queries to conditions in future XPath queries. We can only generalize from conditions in a particular class to other conditions in the same class (i.e., with the same condition annotation).

### 3.2 Structure Annotations

Another problem that we face when designing on-line XML statistics for hidden Web data sources is that the result of an XPath query does not give any information about the part of the XML tree that was navigated to get this result.

For example, consider the XPath query  $//A/B/C$ . Figure 3 shows an XML tree in which the path  $//A/B/C$  occurs a certain number of times with only one  $A$  node and one  $B$  node for all the  $C$  nodes. Figure 4 shows a different XML tree in which the path  $//A/B/C$  occurs the same number of times as in the first XML tree, but with one  $A$  node and one  $B$  node per  $C$  node. Knowing the result of the XPath query  $//A/B/C$  does not help us to distinguish between these two cases.

Our solution to this problem is *not to make any guesses about the structure of the XML tree*. Such guesses would be hard to justify given the limited information about the structure of the tree provided by the XPath queries. Instead, we distinguish between the target tag of an XPath path expression and the tags used for navigating the XML tree to get to this target tag. In an XPath query, say  $//A/B/C$ , we annotate the final tag,  $C$ , with an annotation  $D$ , for *destination*, and the preceding tags,  $A$  and  $B$ , with an annotation  $N$ , for *navigation*. Thus, the XPath query becomes  $//A^N/B^N/C^D$ . We only have selectivity information for destination tags. Navigation tags are needed to get to the destination tag, but we do not have selectivity information

$p_i$	$n_i$	$s_i$
$//A^{NU}/B^{NCA}/C^{DU}$	5	25
$//A^{NU}/B^{NU}/C^{DCS}$	13	67
$//C^{DU}$	2	29
$//F^{NCO}/G^{NCA}/H^{DU}$	2	2
.....	...	...
$//B^{NCS}/C^{DU}$	4	90

Figure 5: An on-line annotated path table

for them. In general, we treat  $A^N$  and  $A^D$  as *distinct tags*. Information about  $A$  as the destination of an XPath query does not help us for XPath queries that use  $A$  for navigation.

The  $N$  or  $D$  annotation does not need to be explicitly represented. In any XPath query, only the final tag gets the  $D$  annotation, while all preceding tags get an  $N$  annotation. Thus, the  $N$  or  $D$  annotation of a tag can be inferred from its position in the path expression. In this paper, we make the  $N$  or  $D$  annotation explicit for clarity of exposition, noting that this does not add any overhead to our algorithms.

We combine the condition annotations and structure annotations for path expression tags. For example, if we classify conditions into “conditions on structure” and “conditions on values,” a tag,  $A$ , is annotated as  $A^{NU}$ ,  $A^{NCS}$ ,  $A^{NCV}$ ,  $A^{DU}$ ,  $A^{DCS}$ , or  $A^{DCV}$ . These annotated tags are treated as distinct tags for the purpose of selectivity estimation. Selectivity information for one does not help us for queries involving another. As an example of path annotation, the XPath path expression  $//A[E/F]/B/C[@a = "val"]$  is annotated as  $//A^{NCS}/B^{NU}/C^{DCV}$ . We call this representation an *annotated path expression*.

## 4. ON-LINE ANNOTATED PATH TABLES

Our approach to building on-line XML statistics for a hidden Web data source is to store the annotated path expressions corresponding to all XPath queries issued to this one data source in a table that we call the *on-line annotated path table*. If we have multiple data sources, we would have one on-line annotated path table per data source. Every entry in an on-line annotated path table corresponds to one annotated path expression. An entry,  $i$ , stores the annotated path expression it represents,  $p_i$ , or a *hash value* of  $p_i$ , the number of observed XPath queries that correspond to this annotated path expression,  $n_i$ , and the *total* result size of all these  $n_i$  queries,  $s_i$  (i.e., the sum of all the individual result sizes). Figure 5 shows an example on-line annotated path table in which the condition annotation represents whether the condition is simple ( $CS$ ), conjunctive ( $CA$ ), or disjunctive ( $CO$ ).

When an XPath query is issued to a hidden Web data source, we observe the actual result size of the query and use it to update and refine the on-line annotated path table. We determine the annotated path expression corresponding to the XPath query. If this annotated path expression is found in the table, its entry is updated (the  $n_i$  value is incremented and the result size of the XPath query is added to the  $s_i$  value). If the annotated path expression is not found in the table, a new entry is created for it.

To estimate the selectivity of an XPath query using an on-line annotated path table, we determine the annotated path expression corresponding to this query and look up this path expression in the table. If the path expression is found

in the table, the estimated selectivity of the XPath query is  $s_i/n_i$ , the average selectivity of all previous executions of XPath queries corresponding to this annotated path expression. Under our assumptions, the result size of a query corresponding to an annotated path expression is a good predictor of the result size of any other query corresponding to the same annotated path expression. If the annotated path expression corresponding to the XPath query whose selectivity is being estimated is not found in the table, we estimate the selectivity to be 0.

An on-line annotated path table collects and aggregates information about the selectivities of XPath queries issued to a hidden Web data source. The path annotations allow us to aggregate information from several queries in one table entry. They also allow us to generalize the information obtained from observed XPath queries to estimate the selectivity of different, previously unseen XPath queries. As more XPath queries are observed, more and more information is added to the table, so the selectivity estimates it provides become more accurate.

To bound the amount of memory consumed by an on-line annotated path table, we need a mechanism to *remove* path expressions from the table. We specify two memory thresholds: a *target threshold*,  $t_1$ , and a *trigger threshold*,  $t_2$ , such that  $t_1 \leq t_2$ . When the table size reaches  $t_2$ , a table summarization process is triggered. The table is summarized until its size drops to  $t_1$  or less.  $t_1$  can be viewed as the available memory budget at which we want the table size to stabilize. However, we allow the table to grow to  $t_2$  so that there is an opportunity for collecting enough information to improve selectivity estimation accuracy. This additional information that is collected also improves the accuracy and stability of the table summarization process.

To summarize an on-line annotated path table, we remove the entries with the *lowest  $s_i$  values*. A low  $s_i$  value for a table entry can mean one of two things. It can mean that the annotated path expression of this entry occurs only infrequently in the virtual XML document representing the hidden Web data source, so the total result size of all XPath queries corresponding to this annotated path expression will be small even if there are many such queries. A low  $s_i$  value for a table entry can also mean that few XPath queries issued to the data source correspond to the annotated path expression for this entry. This information is reflected directly in the  $n_i$  value, but also indirectly in the  $s_i$  value. In all cases, the entry with the low  $s_i$  value is a good candidate for removal because it represents an infrequently occurring path or an infrequently queried path.

We favor removing path expressions that occur infrequently at the data source over ones that occur frequently because it is very important for query optimization to have accurate selectivity estimates for frequent values. These selectivity estimates help us avoid “big mistakes” like using an un-clustered index for a predicate that is not very selective. Accurate selectivity estimates for infrequent values can certainly be important, but they are less important than accurate selectivity estimates for frequent values. This reasoning is reflected in *end-biased histograms* that explicitly store the frequent values of a data distribution, and that are used in commercial database systems such as DB2.

We also favor removing infrequently queried path expressions over frequently queried path expressions because we want the on-line annotated path table to be *workload-aware*.

If the user workload contains certain queries with higher frequency than others, we want to ensure selectivity estimation accuracy for those frequent queries.

When we remove entries with low  $s_i$  values from an on-line annotated path table, we can aggregate the information contained in the removed entries in table entries that correspond to special path expressions that we call *star path expressions*. A star path expression entry stores aggregated selectivity information for multiple path expressions removed from an on-line annotated path table. In this approach, an on-line annotated path table has entries for *two* star path expressions: a path expression  $//*^{DU}$ , and a path expression  $//*^{DC}$ . The entry for the path expression  $//*^{DU}$  contains the total  $n_i$  and  $s_i$  values of all removed entries whose path expressions contain only unconditional navigation steps (i.e., only  $U$  condition annotations). The entry for the path expression  $//*^{DC}$  contains the total  $n_i$  and  $s_i$  values of all removed entries whose path expressions have one or more conditional navigation steps (i.e., some  $C_x$  annotation on one or more tags). We make the distinction between path expressions with conditional and unconditional navigation because of the high impact that conditions have on selectivity. Note that these star path expressions are similar to the star path expressions used in our work on building statistics for static XML data [1]. Another alternative in table summarization is *not* to use star path expressions. In this case, the entries removed from an on-line annotated path table are simply discarded and the information they contain is lost.

## 5. EXPERIMENTAL EVALUATION

Our goal is to build on-line XML statistics for hidden Web data sources that export their responses to user queries in XML. Unfortunately, as mentioned earlier, publicly available hidden Web data sources do not currently export their data in XML, although we can expect them to do so in the near future. As such, we evaluate our proposed statistics using *static XML data*, using the static XML documents in our experiments as proxies for the virtual XML documents that would be queried in the hidden Web.

We present the results of experiments on two real data sets. For a more extensive experimental evaluation on real and synthetic data sets, please refer to [2]. The first data set we use consists of protein sequence data from the SWISS-PROT database [18]. This data set is 141MB in size, and it contains 4,243,031 XML elements. The second real data set consists of bibliographic entries from the DBLP bibliography [8]. This data set is 48MB in size, and it contains 1,399,765 XML elements. In a real deployment of our technique, SWISS-PROT and DBLP would be data *sources* that export an XML view of parts of the data that they store in databases.

The query workloads we use in our experiment consist of 1000 XPath queries each. All queries ask for paths that do occur one or more times in the data. Each query has a random number of navigation steps between 1 and 4. Each step in the generated queries has a condition with probability  $p$ , where  $p$  is a parameter. The conditions we generate consist of one to three condition atoms connected by the logical operators “and” or “or.” The condition atoms are simple conditions on structure and values. 80% of the generated conditions have one condition atom, 10% have two atoms, and 10% have three atoms. The condition atoms are connected

$p$	SWISS-PROT	DBLP
0%	424,129	56,941
10%	352,122	49,580
25%	249,392	36,745
50%	128,030	25,014

**Table 1: Average result sizes of the query workloads**

by “and” with probability 50% and by “or” with probability 50%. Examples of query path expressions generated by our query generation process for the DBLP data set include `//inproceedings[year="1999" and author="Jones"]/booktitle` and `//article/journal[text()="Algorithmica"]`. In our experiments, we use workloads with condition probability  $p = 0\%$ ,  $10\%$ ,  $25\%$ , and  $50\%$ . The average result sizes of the 1000 queries in these four workloads on each of the two data sets are presented in Table 1.

We use the Xalan XPath processor [19] to execute the queries in our workloads and obtain their result sizes. To simulate a sequence of user queries to a hidden Web data source, we start with an empty on-line annotated path table and issue the queries in a workload one by one. For every query, we estimate its selectivity using the on-line annotated path table and we measure estimation accuracy by comparing the estimated and actual selectivity values. After query execution, the actual result size of the query is used to update and refine the path table, making it more accurate.

In the first experiment we present, we investigate the effect of the condition classification and annotation scheme. Table 2 shows the *average absolute error* in selectivity estimation for the workloads with 50% of the navigation steps having conditions (the maximum level of conditions in our workloads) on the two data sets. These absolute errors are best viewed in the context of the average result sizes presented in Table 1. Table 2 presents the selectivity estimation errors using an on-line annotated path table of size 5KB, which is enough to represent all the annotated path expressions in the workloads without needing any summarization. The table presents the average error for the last 800 queries of each workload. We are making the assumption that the first 200 queries are *training* queries while the last 800 queries are *validation* queries, and we are showing the average error for the validation queries. The table presents the errors using three different condition classification and annotation schemes: classifying conditions into conditions on structure and conditions on values; classifying conditions into simple, conjunctive, and disjunctive; and classifying tags into tags with conditions and tags without conditions (i.e., grouping all conditions into one class).

Table 2 shows that classifying conditions into simple, conjunctive, and disjunctive has an advantage over the other two schemes, but only a slight advantage that does not justify its extra complexity. As such, we conclude that it is best to use the simplest scheme of grouping all conditions into one class and annotating the tags in the XPath steps with either  $U$  or  $C$ , for *unconditional* or *conditional*. We use this condition annotation scheme for all our other experiments. In addition to being the simplest scheme with the lowest cost, this scheme allows for the most generalization from observed XPath queries to future XPath queries.

The second experiment we present investigates the convergence and accuracy of the selectivity estimates provided

by on-line annotated path tables. Figure 6 shows the average absolute error in selectivity estimation for the four query workloads on the two data sets. The errors are shown for queries grouped in groups of 100 queries each. The on-line annotated path tables use a target memory threshold,  $t_1$ , of 500 bytes, and a trigger threshold,  $t_2$ , of 1000 bytes. When summarizing the on-line annotated path tables, we use star path expressions to represent removed entries.

The figure shows that on-line annotated path tables have good convergence properties and can “learn” the data distribution fast. The figure also shows that as  $p$  increases and the number of conditions in the query path expressions increases, the effect of the assumption that conditions of a particular form have a uniform effect on selectivity also increases. This leads to an increase in selectivity estimation error. However, the error remains adequately low.

In the third experiment we present, we compare on-line annotated path tables to the static XML statistics that we proposed in [1]. In that paper, we identified two types of static XML statistics as winners among several techniques: *path trees with global-\* summarization*, and *Markov tables with  $m = 2$  and suffix-\* summarization* (see [1] for details).

We compare the on-line annotated path tables that we propose in this paper to these two kinds of static XML statistics. Since the static XML statistics can only handle navigation based on the structure of the XML data and cannot handle conditions in the query path expressions, we only compare them to on-line statistics for workloads with *no conditions* ( $p = 0\%$ ).

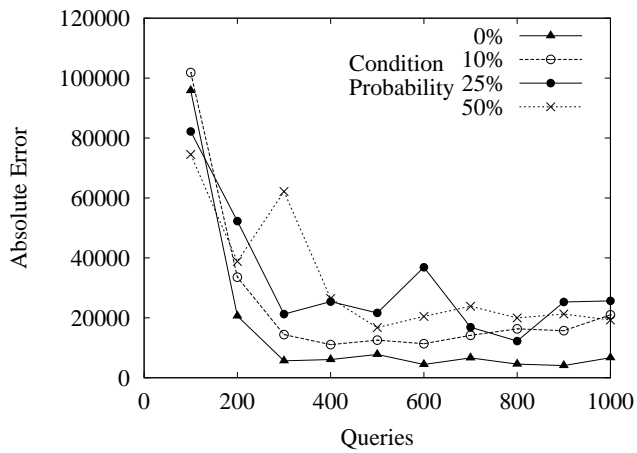
Figure 7 shows the selectivity estimation errors for workloads with no conditions using path trees and Markov tables, as well as using on-line annotated path tables. As in Table 2, the errors are shown for the last 800 queries in each workload (the validation queries) using different memory allocations. For static statistics, the memory allocations shown on the  $x$ -axis are the total number of bytes given to the statistics. For on-line statistics, the memory allocations shown are the target threshold,  $t_1$ . The trigger threshold,  $t_2$ , is set to  $2 * t_1$ . The figure shows that on-line XML statistics are comparable in performance to static XML statistics, and sometimes even better. The figure also shows that the estimation error decreases as the available memory increases.

On-line annotated path tables are built based only on observing user queries and their result sizes. This is much more limited information than is available for static XML statistics, which are built by reading the entire XML data set and processing it as needed. We expect the static statistics built using full information to be more accurate than the on-line statistics built using limited information. This is what we see in Figure 7(a) and in small memory allocations in Figure 7(b). However, the good news from these figures is that the on-line statistics are comparable in accuracy to the static statistics. Thus, even though we *cannot* use static XML statistics for hidden Web data sources because we do not have access to the data, this experiment shows that on-line XML statistics, the only alternative we *can* use, are not much less accurate.

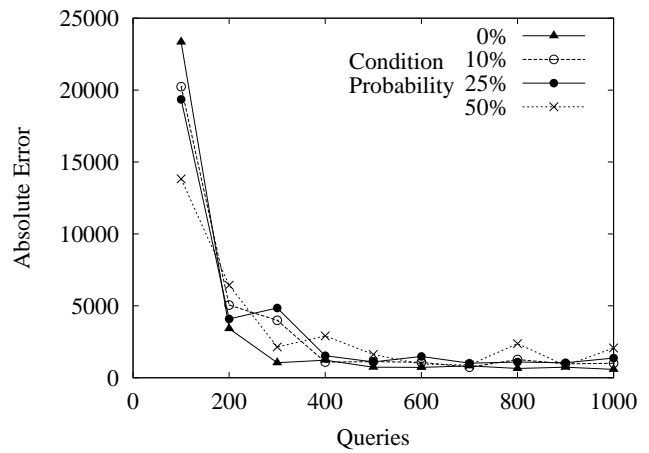
The surprising result that we see in Figure 7(b) is that on-line XML statistics can be *more accurate* than static XML statistics. This is because on-line XML statistics are *workload aware*. On-line XML statistics try to retain information about paths in the data *that are queried by the user*. Static XML statistics, even though they have access to more

Data Set	Structure / Values	Simple / Conjunctive / Disjunctive	Any Condition / No Condition
SWISS-PROT	23,327		23,327
DBLP	1,591		1,591

Table 2: Effect of the condition classification method

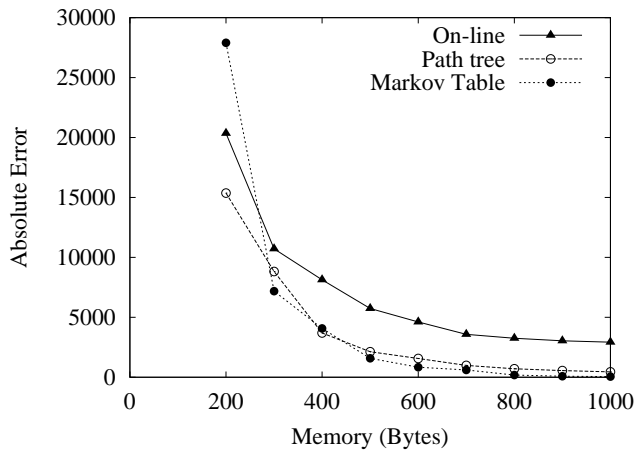


(a) SWISS-PROT

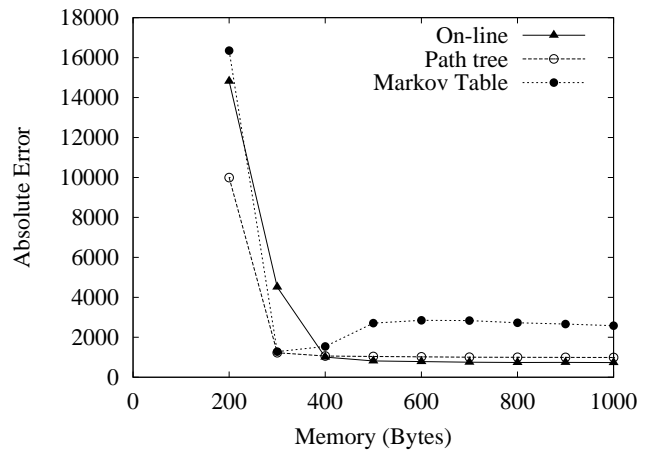


(b) DBLP

Figure 6: Convergence and estimation accuracy for the two data sets



(a) SWISS-PROT



(b) DBLP

Figure 7: Comparison to static XML statistics for the two data sets

information, summarize the data without considering user queries. Thus, they may discard some information during summarization that, while not significant from the point of view of capturing a data distribution, is frequently queried by the user. If the on-line statistics keep this information, they can be more accurate than the static statistics.

## 6. CONCLUSIONS

We propose a novel type of XML statistics for hidden Web data sources that we call *on-line annotated path tables*. An on-line annotated path table for a hidden Web data source stores the XPath query path expressions that were issued to this data source in a more generalized form known as *annotated path expressions*. The table also stores aggregate information about the result sizes of the queries corresponding to these annotated path expressions. This information can be leveraged to estimate the selectivity of subsequent user queries, even if these queries are seen for the first time. A summarization algorithm ensures that the amount of memory used by the table remains bounded.

We see several directions for future work. In this paper, we assume that queries to a hidden Web data source are XPath selections from a virtual XML document representing the data at this source. This model of querying hidden Web data sources is easy to incorporate into XML query processors, and it is general and expressive enough to handle current hidden Web interfaces. However, it would be interesting to investigate other models for querying hidden Web data sources, and to determine the impact of these models on query optimization and processing and on statistics gathering. Developing more elaborate techniques for building and maintaining on-line annotated path tables is another possible area of future work. This includes, for example, more elaborate techniques for handling XPath conditions. It also includes improved table summarization methods, possibly relying on the *recency* of queries, in addition to their frequency, to determine which entries in the on-line annotated path table to remove. Improving on-line annotated path tables can also include developing techniques for detecting changes in the data distribution at the Web sources and reflecting these changes in the tables. Another possible area for future work is developing robust techniques for inferring information about the structure of the XML tree based on the queries in the workload and their results. Finally, it may be possible to utilize semantic knowledge or schema knowledge to construct or refine statistics for hidden Web data sources.

## 7. REFERENCES

- [1] A. Aboulnaga, A. R. Alameldeen, and J. F. Naughton. Estimating the selectivity of XML path expressions for Internet scale applications. In *Proc. Int. Conf. on Very Large Data Bases*, pages 591–600, Rome, Italy, Sept. 2001.
- [2] A. Aboulnaga and J. F. Naughton. Towards building XML statistics for the hidden web. Technical Report TR-CS-03-1477, Computer Sciences Department, University of Wisconsin - Madison, Apr. 2003.
- [3] S. Adali, K. S. Candan, Y. Papakonstantinou, and V. Subrahmanian. Query caching and optimization in distributed mediator systems. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 137–148, Montreal, Canada, June 1996.
- [4] A. Berglund, S. Boag, D. Chamberlin, M. F. Fernandez, M. Kay, J. Robie, and J. Siméon (eds.). XML path language (XPath) 2.0. W3C Working Draft available at <http://www.w3.org/TR/xpath20/>, May 2003.
- [5] S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, and J. Siméon (eds.). XQuery 1.0: An XML query language. W3C Working Draft available at <http://www.w3.org/TR/xquery/>, May 2003.
- [6] The deep web: Surfacing hidden value. White paper available from <http://www.brightplanet.com/>.
- [7] Z. Chen, H. Jagadish, F. Korn, N. Koudas, S. Muthukrishnan, R. Ng, and D. Srivastava. Counting twig matches in a tree. In *Proc. IEEE Int. Conf. on Data Engineering*, pages 595–604, Heidelberg, Germany, Apr. 2001.
- [8] DBLP Bibliography. <http://www.informatik.uni-trier.de/~ley/db/>.
- [9] J. Freire, J. R. Haritsa, M. Ramanath, P. Roy, and J. Siméon. StatiX: Making XML count. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 181–191, Madison, Wisconsin, June 2002.
- [10] Z. G. Ives, D. Florescu, M. Friedman, A. Y. Levy, and D. S. Weld. An adaptive query execution system for data integration. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 299–310, Philadelphia, Pennsylvania, June 1999.
- [11] L. Lim, M. Wang, S. Padmanabhan, J. S. Vitter, and R. Parr. XPathLearner: An on-line self-tuning Markov histogram for XML path selectivity estimation. In *Proc. Int. Conf. on Very Large Data Bases*, pages 442–453, Hong Kong, China, Aug. 2002.
- [12] H. Naacke, G. Gardarin, and A. Tomasic. Leveraging mediator cost models with heterogeneous data sources. In *Proc. IEEE Int. Conf. on Data Engineering*, pages 351–360, Orlando, Florida, Feb. 1998.
- [13] J. Naughton, D. DeWitt, D. Maier, et al. The Niagara Internet query system. *IEEE Data Engineering Bulletin*, 24(2):27–33, June 2001.
- [14] N. Polyzotis and M. Garofalakis. Statistical synopses for graph-structured XML databases. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 358–369, Madison, Wisconsin, June 2002.
- [15] N. Polyzotis and M. Garofalakis. Structure and value synopses for XML data graphs. In *Proc. Int. Conf. on Very Large Data Bases*, pages 466–477, Hong Kong, China, Aug. 2002.
- [16] S. Raghavan and H. Garcia-Molina. Crawling the hidden web. In *Proc. Int. Conf. on Very Large Data Bases*, pages 129–138, Rome, Italy, Sept. 2001.
- [17] M. T. Roth, F. Ozcan, and L. M. Haas. Cost models DO matter: Providing cost information for diverse data sources in a federated system. In *Proc. Int. Conf. on Very Large Data Bases*, pages 599–610, Sept. 1999.
- [18] SWISS-PROT protein sequence database. <http://www.expasy.ch/sprot/>.
- [19] Xalan-C++. <http://xml.apache.org/xalan-c/>.
- [20] L. Xyleme. A dynamic warehouse for XML data of the web. *IEEE Data Engineering Bulletin*, 24(2):40–47, June 2001.