Paper ID 604

Building XML Statistics for the Hidden Web

Ashraf Aboulnaga
Jeffrey F. Naughton
Computer Sciences Department
University of Wisconsin - Madison
1210 West Dayton Street
Madison, WI 53706

{ashraf,naughton}@cs.wisc.edu

(No authors on VLDB 2002 program committee)

Contact Author: Ashraf Aboulnaga
E-mail: ashraf@cs.wisc.edu
Phone: ++1-608-262-6623
Fax: ++1-608-262-9777
Reference Number: 604
Topic Area: Core Database Technology
Category: Research
Relevant Topics: Semi-structured Data, XML — Internet and the WWW — Optimization and Performance

Building XML Statistics for the Hidden Web

Ashraf Aboulnaga Jeffrey F. Naughton

University of Wisconsin - Madison
{ashraf,naughton}@cs.wisc.edu

Abstract

There is currently a lot of interest in developing Internet query processors that can pose elaborate queries on XML data on the Web. Such query processors can query data sources that have static XML files, but they should also be able to query *"hidden Web"* data sources that export an XML view of data stored in a database. To optimize queries that involve these hidden Web data sources, we need to have XML statistics that can be used to estimate the selectivity of queries posed to these sources. Since we can only access the data at a hidden Web data source by issuing queries, we need to develop *on-line* XML statistics that are built by observing queries to a hidden Web data source and their result sizes.

In this paper, we assume that queries to a hidden Web data source are XPath selections from a virtual XML document that represents all the data at this source. We observe the user XPath queries to the data source and convert them to a more abstract and generalized form that we call *annotated path expressions*. We describe an on-line statistics structure that stores such annotated path expressions and information about their selectivity for use in estimating the selectivity of future XPath queries. We experimentally demonstrate the convergence and accuracy of our proposed on-line statistics using real and synthetic XML data sets.

1 Introduction

There is currently a lot of interest in developing Internet query processors that can "query the Web." Such query processors would retrieve and integrate data from multiple Web sources. They would provide the user with high-quality information that is much more useful than that which can be obtained today, even with advanced search engines.

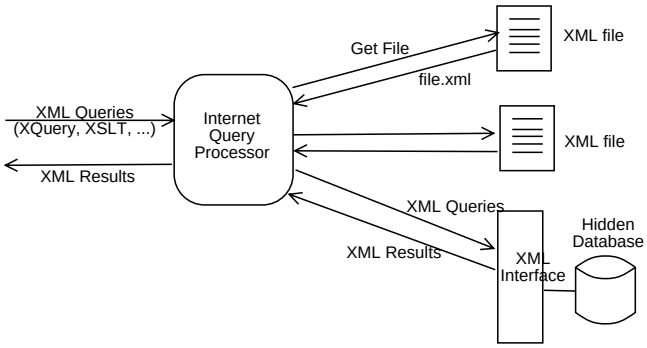

Figure 1: Querying the Web using XML

The emergence of XML as a standard data representation format for Web data is a key factor in facilitating the development of such Internet query processors. XML provides a common format for all Web sources to export their data, so Internet query processors can be built assuming that all the data that they query will be in XML. Examples of systems that query XML data over the Internet include Niagara [NDM+01] and Xyleme [Xyl01].

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 [RGM01, Bright]. This portion of the Web is known as the *hidden Web*. Sometimes it is also referred to as the *deep Web*.

XML has gained almost universal acceptance as the standard format for interchanging data between data sources on the Internet [FLM98]; even more so than as a format for storing static data. As such, we can expect that hidden Web data sources will export the data they produce in response to user queries in XML format. Therefore, it should be possible – and highly desirable – for Internet query processors like Niagara to query hidden Web data using the same XML query processing framework that they use to query the static Web, as shown in Figure 1. Optimizing XML queries over hidden Web data sources requires *statistics* about these sources. Providing such statistics is the focus of this paper.

**Proceedings of the 28th VLDB Conference,
Hong Kong, China, 2002**

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 [Bright]. Examples of hidden Web data sources include the FactFinder database of census information from the U.S. Census Bureau [Census] and the EDGAR database of company financial statements from the Securities and Exchange Commission [EDGAR].

These data sources do not currently present their responses to user queries as as XML, but rather as HTML. However, as XML gets deployed on a wider scale, we can expect that many hidden Web data sources will start exporting their data as XML. Furthermore, it is often possible to build *wrappers* around Web sites that can present an XML view of the HTML data at these sites [PGMW95]. As such, even though XML data from hidden Web data sources is not widely available yet, it should be useful to include the ability of dealing with such data in Internet query processors like Niagara.

1.1 A Motivating Example

As an example of the queries that can be made possible by XML based Internet query processors querying hidden Web data sources, consider the query in Figure 2 expressed in the XQuery language [CFR+01]. This is a join query that asks for price quotes under $25,000 from car dealers in Madison for year 2002 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, albeit in HTML not XML. However, as mentioned above, it would be reasonable to expect that this information may be available in XML in the near future, thereby making such a query feasible.

The query in Figure 2 uses XPath [CD99] path expressions to query the hidden Web data sources. Each path expression specifies a navigation through the structure of the XML data based on a sequence of tags, with possible conditions at each step to filter out some of the XML elements encountered at this step. For example, the path expression `//newcar/quote[city="Madison"]` specifies finding XML elements with `newcar` tags anywhere in the specified XML document, and finding all XML elements directly contained within these `newcar` elements that have a `quote` tag. The condition between braces specifies that we should only return `quote` elements that directly contain an element with a tag `city` whose value is "Madison." The XQuery language uses XPath path expressions to navigate through XML data, and we assume that queries to hidden Web data sources are in the form of XPath path expressions (we present more details in Section 3.1).

In this paper, we focus on the problem of estimating the selectivity of XPath path expressions issued to hidden Web data sources. This is required for optimizing queries like the one in Figure 2. Estimating the selectivity of these XPath path expressions requires *statistics* about the data at the hidden Web data sources. More accurately, it requires statistics about the *XML view* of the data at these sources.

1.2 On-line XML Statistics for the Hidden Web

The typical approach for building statistics for relational or XML data is to scan the entire data and summarize it in a structure that occupies a small amount of memory. This does not work for XML data on the hidden Web because *we do not have access to the entire data*. We only have access to queries on this data and their results.

Most hidden Web data is stored in relational databases, and sometimes in document databases. It 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.

We develop *on-line XML statistics* for this environment. These statistics are constructed by *observing user queries to hidden Web data sources and their results*. The statistics are built *per hidden Web data source*. They use information from past queries to a data source to estimate the selectivity of *different* future queries to this data source. We do not require any cooperation from the hidden Web data sources in building these statistics; the statistics are based solely on feedback from user queries. To reduce the construction overhead and the complexity of the statistics, we assume that we can only use 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.

The problem we have described is daunting in the extreme: we are asking for statistics about an enormous, complex, opaque data set; we cannot even view this data set, being able only to observe it indirectly as queries and their results hint at its structure and size. In such a situation, it is unrealistic to expect solutions of the same quality as those that have been developed for the highly constrained environment of query optimization for relational data or static XML data. That is not our goal. Instead, we seek to take the first step toward developing techniques that yield statistics that are substantially better than having no statistics at all for hidden Web data sources. The information that these statistics provide should, hopefully, be useful in optimizing queries over the hidden Web.

The rest of this paper is organized as follows. Section 2 presents an overview of related work. Section 3 contains a detailed definition of the problem that we are addressing. Section 4 introduces *path annotations*, which we use in our on-line statistics. Section 5 describes these statistics, which we call *on-line annotated path tables*. Section 6 presents an experimental evaluation of our proposed technique. Section 7 contains concluding remarks.

```
FOR $r IN document("http://www.nhtsa.gov/")//safety/car[year=2002 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 2: An example query in the XQuery language

2 Related Work

Statistics for *static* XML data have been proposed by Chen et al. in [CJK+01] and by us in [AAN01]. The techniques in [CJK+01] build statistics that are used to estimate the selectivity of *twig queries*. Twig queries are branching path expressions which can include conditions on the values of the leaf nodes of the branches. The statistics we propose in [AAN01] 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. The techniques in both these papers are not applicable to the hidden Web, because they require the entire XML data to be read to construct the statistics.

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 [IFF+99], Garlic [ROH99], or HERMES [ACPS96]. 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 [NGT98, ROH99]. 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.

Another approach is to design the data integration system to allow for run-time re-optimization of queries [IFF+99]. 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 initial plan. Starting with this good plan, 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 [ACPS96]. 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 Problem Definition

3.1 Our Model for Hidden Web Queries

We view a hidden Web data source as a *virtual XML document*. This virtual XML document represents all possible XML query results that the data source can produce in response to user queries. If the data source can present different XML views of the same data, each of these views is considered to be a separate part of the virtual XML document. For example, the car safety rating data source we accessed in the query in Figure 2 may be able to present an XML view of the safety ratings of cars grouped by make, and another XML view of the different cars grouped by safety rating. Each of these two XML views of the data would be a different part of the virtual XML document representing the data source, and the car safety data would be replicated in both these parts.

The virtual XML document representing a hidden Web data source can be very large, since it contains the answers to *all* possible user queries that the data source can support, with the data replicated possibly many times. However, this is not a problem because this virtual XML document is *never materialized*. Only parts of this document are ever materialized, and only in response to user queries.

We assume that queries to a hidden Web data source are in the form of *XPath path expressions* [CD99] that select parts of the virtual XML document representing this data source. XPath is the standard path expression language for selecting parts of an XML document based on structure and content. It is a powerful language that can express many kinds of selections, including most queries that can be input using current HTML forms. Furthermore, XPath is used in XML query standards such as XQuery [CFR+01] and XSLT [Clark99].

We consider XPath path expressions of the form $//a_1/a_2/\cdots/a_n$. Each step, a_i, of the 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. Such queries find element nodes with tag name t_1 anywhere in the XML tree representing the document. If there is a condition c_1, element nodes that do not satisfy it are filtered out. From the remaining nodes, the queries navigate down to all t_2 children, then down to all t_3 children, and so on until they reach t_n element nodes. At each step, if there is a condition, nodes that do not satisfy it are filtered out. Our goal is to estimate the number of t_n nodes that are reached by this navigation. Examples of XPath queries that we consider are `//safety/car[make="Saturn" and year=2002]/rating`, and `//chapter[@title="Introduction"]/section[1]/paragraphs`.

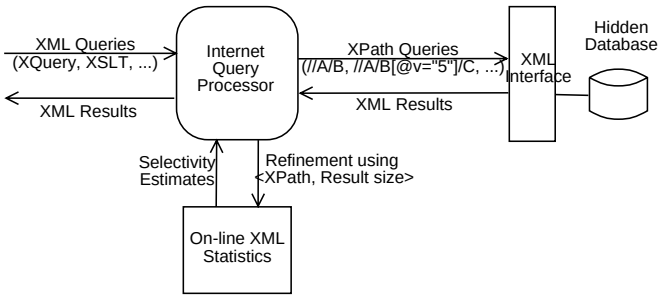

Figure 3: On-line XML statistics

3.2 Problem Definition

We consider the following setting: As part of answering user queries, an Internet query processor issues queries to a hidden Web data source. These queries are in the form of XPath path expressions that select parts of the virtual XML document representing this source. The XPath queries are executed by the data source, and their results (the XML elements they select) are returned to the Internet query processor, where they are used to answer the user queries.

We observe the XPath queries issued to a hidden Web data source and their result sizes (the number of XML elements they return). Our objective is to use these observations to construct on-line XML statistics for the hidden Web data source (Figure 3). These 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. Furthermore, there should be a mechanism for bounding the amount of memory consumed by the statistics to any given value.

4 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 an XPath query whose selectivity is being estimated is identical to a query that was previously issued, we would find this query and its exact result size in the query cache. We would, therefore, have a fully accurate selectivity estimate for this query, assuming a read-only data source. However, for this technique to work, it must cache *every* query and its result size. If the query workload consists of a large number of queries, the statistics data structure will grow unacceptably large. Furthermore, this simplistic solution is of no use for XPath queries that are seen for the first time. Our on-line XML statistics must fit in a small amount of memory, and they must be able to generalize the information obtained from previously seen XPath queries to estimate the selectivity of future XPath queries that are seen for the first time. To allow for this, we use *path annotations*.

We convert XPath path expressions into more abstract and general *annotated path expressions*, and we use these annotated path expressions for selectivity estimation. An annotated path expression represents *all* XPath path expressions that have a particular form, so it provides a degree of summarization. The annotated path expression for one or more observed XPath queries can be used to estimate the selectivity of *different* future XPath queries that correspond to this annotated path expression. 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. Next, we describe the details of two types of path annotations: *condition annotations* and *structure annotations*.

4.1 Condition Annotations

We consider XPath path expressions of the form $//a_1/a_2/\cdots/a_n$, where each step, a_i, of the path expression can be of the form $t_i[c_i]$, where t_i is a tag name and c_i is an arbitrary condition. Allowing arbitrary conditions in the XPath path expressions creates the problem of how to deal with these conditions in the on-line XML statistics.

On the one hand, to be realistic, 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 selections from the virtual XML document corresponding to a hidden Web data source. 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 2002 Saturns" (`//safety/car[make="Saturn" and year=2002]/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 XPath query `//safety/car[make="Saturn" and year=2002]/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=2002]`. This is generally true of all conditions in typical XPath steps: their effect on selectivity is large and cannot be ignored. But at the same time, the effect of conditions on selectivity cannot be isolated as for static XML data.

When building statistics for static XML data, we can – conceptually, at least – traverse the data and count the number of XML elements that are reachable by the path `//safety/car`, and the number of these elements that satisfy the condition `[make="Saturn" and year=2002]`. Thus, we can conceptually isolate the effect of the condition on selectivity. However, when building statistics for hidden Web data sources, we can only observe the entire XPath queries and their result sizes, with no opportunity for isolating single conditions or navigation steps. Furthermore, a condition on a tag in a particular XPath query can have a different effect on selectivity from the same condition on the same tag in a different XPath query. For example, the condition on tag C in the XPath query $//B/C[cond_1]$ can have a different ef-

fect on selectivity from the same condition in the query $//A/B[cond_2]/C[cond_1]$.

Our solution to the problem of handling conditions in the XPath queries when building on-line XML statistics is to make the assumption that *conditions have a uniform effect on selectivity*. This means that a condition on a tag in an XPath path expression has the same effect on selectivity as any other condition on this tag. To use an XPath path expression in our statistics, we *annotate* every tag in this path expression with U or C depending on whether or not this tag has a condition. If a tag, A, has no condition, we annotate it with a U, for *unconditional*. If the tag, A, has a condition (i.e., the XPath step is $A[cond]$, for some condition $cond$), we annotate the tag, A, with a C, for *conditional*. Thus, for selectivity estimation purposes, a tag, A, becomes either A^U or A^C. We treat A^U and A^C as *distinct tags*.

Since we assume that conditions have a uniform effect on selectivity, selectivity information obtained from A^C tags can be used to estimate the selectivity of subsequent A^C tags, regardless of the condition that caused the C annotation. However, since we treat A^U and A^C as distinct tags, information about A^U *cannot* be used to estimate the selectivity of A^C, and vice versa.

The assumption that conditions have a uniform effect on selectivity is admittedly a strong one, especially since we allow arbitrarily complex conditions. For example, this assumption implies that the conditions in the two XPath queries `//safety/car[make="Saturn" and year=2002]` and `//safety/car[body_style="sedan"]` will be considered to have the same effect on selectivity. However, this assumption reduces the difficult problem of handling conditions in the XPath queries to a tractable problem for which we propose a simple and uniform solution. Furthermore, many hidden Web data sources allow only very stylized forms of queries, such as those corresponding to HTML forms. In this case, all conditions will have a similar form, so the assumption that they all have the same effect on selectivity may well hold. Also, our experiments demonstrate that our statistics converge to an adequate accuracy, even with this assumption.

4.2 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 4 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 5 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. The result of this query gives us

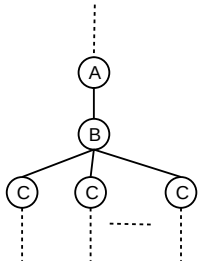

Figure 4: Minimum number of ancestors for C

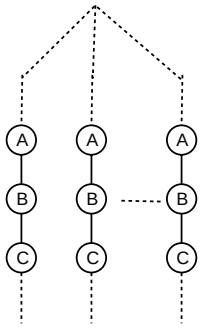

Figure 5: Maximum number of ancestors for C

information about the number of C nodes, not the number of A or B nodes. The result of an XPath path expression gives no information about its prefixes. This is different from the case of static XML data, in which we have full access to the XML tree and can explore it any way we want to get the required information.

In our on-line XML statistics, *we do not 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 XPath queries. Instead, we only consider the selectivity of *full* XPath path expressions.

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 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.

We combine the condition annotations and structure annotations for path expression tags. Thus, a tag, A, gets annotated as A^{NU}, A^{NC}, A^{DU}, or A^{DC}. These annotated tags are treated as four distinct tags. Selectivity information for one does not help us for queries involving another. As an example of path annotation, the XPath path expression $//A[2]/B/C[@a = "val"]$ becomes $//A^{NC}/B^{NU}/C^{DC}$. We call this an *annotated path expression*.

p_i	n_i	s_i
$//A^{NU}/B^{NC}/C^{DU}$	5	25
$//A^{NC}/B^{NC}/C^{DU}$	7	90
$//A^{NU}/B^{NU}/C^{DC}$	13	67
$//C^{DU}$	2	19
$//F^{NC}/G^{NC}/H^{DU}$	2	2
.....	...	...
$//B^{NC}/C^{DU}$	4	90

Figure 6: An on-line annotated path table

5 On-line Annotated Path Tables

In this section, we describe a novel kind of on-line XML statistics for hidden Web data sources, which we call *on-line annotated path tables*. An on-line annotated path table stores information for *one* hidden Web data source. The table stores the annotated path expressions corresponding to the XPath queries issued to this data source and information about their selectivities. This information is used to estimate the selectivity of XPath queries subsequently issued to the data source, and the table is updated with feedback information from the execution of these queries.

Every entry in an on-line annotated path table corresponds to one annotated path expression. The entry stores information about all previously executed XPath queries that correspond to this annotated path expression. In particular, an entry, i, stores the annotated path expression it represents, 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). To reduce the amount of memory required by the table, the entries can store *hash values* of the annotated path expressions they represent instead of the full path expressions themselves. Using this optimization, each entry requires 3 integers (12 bytes), one for each of $hash(p_i)$, n_i, and s_i.

Figure 6 shows an example on-line annotated path table. For clarity of exposition, the figure shows the table storing actual annotated path expressions. In our implementation of on-line annotated path tables, we do not store the full annotated path expressions but only their hash values.

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 corresponding to this data source. First, we determine the annotated path expression corresponding to the XPath query. Next, we look up this annotated path expression in the on-line annotated path table. If the annotated path expression is found in the table, the corresponding n_i value is incremented by 1, and the result size of the XPath query is added to the corresponding s_i value. If the annotated path expression is not found in the table, a new entry is created for this path expression, with n_i equal to 1 and s_i equal to the result size of the XPath query.

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. This is 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 can be used as a predictor of the result size of any other query corresponding to the same annotated path expression. The selectivity estimate s_i/n_i reflects information about *all* previous queries that correspond to the same annotated path expression as the current query.

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. In this case, there is no information about previous queries that correspond to the same annotated path expression as the current query.

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.

5.1 Table Summarization

The previous section describes how we *add* annotated path expressions to an on-line annotated path table. If we only add path expressions to the table, it will grow indefinitely. This is clearly unacceptable. Hence, we need a mechanism to *remove* path expressions from the table so that we can bound the amount of memory it consumes.

When building statistics for static XML data, a common approach is to build the statistics completely, with no restrictions on the amount of memory that they consume, and then to summarize the statistics so that they fit in the available memory [AAN01, CJK+01]. For our on-line statistics, there is no notion of the construction of the statistics being completed. Information is continuously added to the statistics when user queries are issued. Thus, we cannot build the statistics "to completion" and then summarize them.

Instead, to bound the amount of memory consumed by an on-line annotated path 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 of the table summarization process. Allowing the table to grow to t_2 also adds stability to the table summarization process.

The trigger threshold, t_2, can have any value greater than or equal to the target threshold, t_1. The greater the difference $t_2 - t_1$, the fewer times the table has to be summarized. Fewer summarizations can potentially mean greater accuracy for the table. In this paper, we set $t_2 = \alpha t_1$, where $\alpha \geq 1$ is a parameter of the table construction process. Another alternative could to be to set $t_2 = \min(\alpha t_1, \beta)$, where β is the maximum memory size to which we are willing to allow the table to grow.

To summarize an on-line annotated path table when its size reaches the trigger threshold, t_2, we remove from the table 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. In both these 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.

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*. An on-line annotated path table can have entries for *two* star path expressions: a path expression $//*^{DU}$, and a path expression $//*^{DC}$. The entry for path expression $//*^{DU}$ contains the total n_i and s_i values of all removed entries whose path expressions have only U annotations on their tags (i.e., path expressions with only unconditional navigation). The entry for path expression $//*^{DC}$ contains the total n_i and s_i values of removed entries whose path expressions have a C annotation on some tag or tags (i.e., path expressions with some conditional navigation).

We make the distinction between path expressions with conditional and unconditional navigation because of the high impact that conditions have on selectivity. The table entries for the star path expressions represent the information contained in the removed table entries at a coarser granularity. They are similar to the star paths we used in our work on building statistics for static XML data [AAN01].

Another alternative is *not* to use star paths. In this case, the entries removed from an on-line annotated path table are simply discarded and the information they contain is lost.

6 Experimental Evaluation

6.1 Experimental Setup

6.1.1 Data Sets

Our goal in this paper 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.

To evaluate our on-line annotated path tables on a particular static XML document, we issue a sequence of XPath queries on this document. We build the on-line annotated path table for this XML document by observing these queries and their result sizes. In the process, we also use the table for estimating the selectivity of the queries, just as we would do for a hidden Web data source. Nowhere in our experiments do we assume that we have access to the XML document. We only use the sequence of XPath queries on the document and their result sizes. The scenario would be exactly the same for a true hidden Web data source, except that instead of the queries being on a *static* XML document, they would be on the *virtual* XML document representing the data source. Thus, we are 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 and one synthetic data set. The first real data set consists of protein sequence data from the SWISS-PROT database [SPROT]. 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 [DBLP]. 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 relational or other databases.

The synthetic data set we use in our experiments is generated using the XML data generator described in [ANZ01]. The tree representing the structure of this data set has 8 levels, and the nodes of this tree (which correspond to the XML elements) have frequencies that follow a Zipfian distribution with parameter $z = 1$ [Zipf49]. The data set is 17MB in size, and it contains 1,000,000 XML elements. The values stored within the XML elements are a total of 1,000,000 text words generated from a Zipfian distribution with 10,000 distinct words and $z = 1$. More details about this data set can be found in Appendix A.

6.1.2 Query Workloads

The query workloads we use in our experiments 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.

To generate a query in our workloads, we choose a random node from the XML tree of the data set and make it the destination node of the query path expression. This node can be an internal node or a leaf node of the XML tree. If the length of the query path expression to be generated is greater than 1, the ancestors of this destination node become navigation steps in the query path expression. This fully specifies the *navigation* component of the query path expression (the sequence of tags).

To control the generation of *conditions* in the query path expressions, we specify a parameter, p, of the query gener-

p	SWISS-PROT	DBLP	Synth
0%	424,129	56,941	17,753
10%	352,122	49,580	15,986
25%	249,392	36,745	11,324
50%	128,030	25,014	6,737

Table 1: Average result sizes of the query workloads

ation process which we call the *condition probability*. This parameter is the probability of any step in the generated XPath query path expressions having a condition on its tag. For every step of a generated query path expression, we flip a coin with success probability p to determine whether or not this step includes a condition.

The conditions we generate for XPath steps consist of one to three *condition atoms* connected by the logical operators "and" or "or." 80% of the generated conditions have one condition atom, 10% have two atoms, and 10% have three atoms. The condition atoms are connected by "and" with probability 50% and by "or" with probability 50%.

To generate a condition atom for an XPath step corresponding to a particular node in the XML tree, we check if this node is an internal node or a leaf node. If it is a leaf node, the generated condition atom is `text()="val"`, where *val* is the string value contained in this leaf node. This condition atom specifies that we only want leaf nodes that contain this particular string value. If the node in the XML tree that corresponds to the XPath step for which we want to generate a condition atom is an internal node, the generated condition atom depends on whether or not this node has children that contain string values. If the node has one or more children that contain string values, we choose one of these children at random, say `C`, and we generate the condition atom `C="val"`, where *val* is the string value contained in `C`. If the node has no children that contain string values, we choose one of its children at random, say `D`, and we generate the condition atom `D`. This specifies a condition based solely on the structure of the XML data.

Our query generation process generates diverse workloads. Some queries ask for internal nodes and some ask for leaf nodes. Some have conditions and some do not. The conditions in the queries are based on both structure and values, and some of them are atomic while others are complex. Examples of XPath 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 three data sets are presented in Table 1. We use the Xalan XPath processor [Xalan] 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 the information in the path table, thereby making it more accurate.

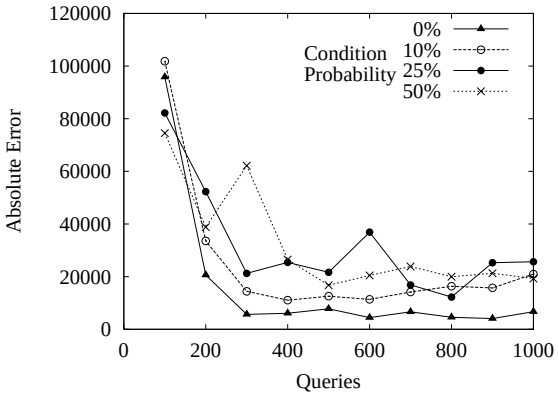

Figure 7: Convergence (SWISS-PROT)

6.1.3 Error Metric

The error metric we use to measure selectivity estimation accuracy is the *average absolute error*. The average absolute error for a set of N queries is defined as $\frac{1}{N}\sum_{i=1}^{N}|est - act|$, where est is the estimated selectivity and act is the actual selectivity.

Except for the convergence experiment in Section 6.2, we evaluate our techniques based on the average estimation error of the *last 800 queries* of each workload. We assume that the first 200 queries are *training* queries and the remaining 800 queries are *validation* queries.

The errors we present may be better viewed in the context of the average result sizes of the queries in our query workloads (Table 1).

6.1.4 Default Parameters

Our default memory allocation for on-line statistics is to use a target threshold, t_1, of 500 bytes, and a trigger threshold, $t_2 = \alpha t_1$. Our default value for α is $\alpha = 2$, so $t2$ is 1000 bytes. We use a small memory allocation because the simple information reflected in on-line annotated path tables does not require a lot of memory to store, especially for 1000 queries. Furthermore, our statistics represent information about a single hidden Web data source, and an Internet query processor may deal with thousands of such sources. As such, we have to be very parsimonious with our memory allocation.

When summarizing on-line annotated path tables, our default is to use star path expressions to represent the removed table entries.

Unless otherwise specified, we use workloads with the condition probability parameter, p, set to 25%.

6.2 Convergence

In this section, we study the convergence of the selectivity estimates provided by on-line annotated path tables. We

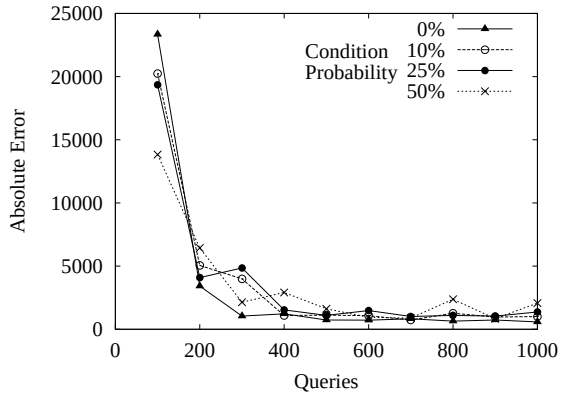

Figure 8: Convergence (DBLP)

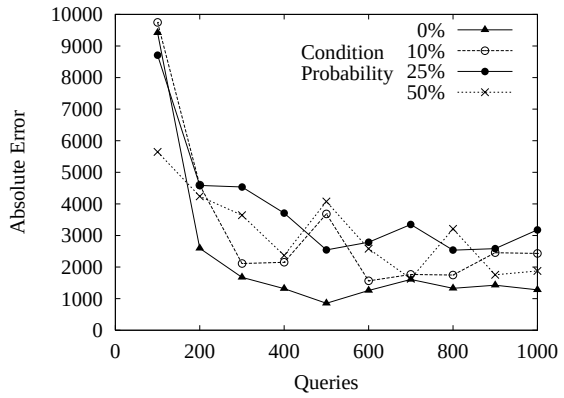

Figure 9: Convergence (synthetic)

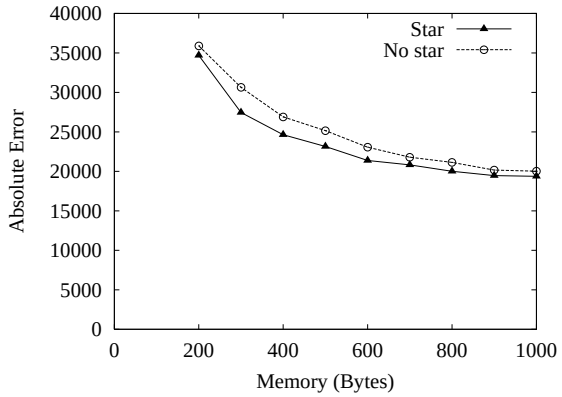

Figure 10: Memory and summarization (SWISS-PROT)

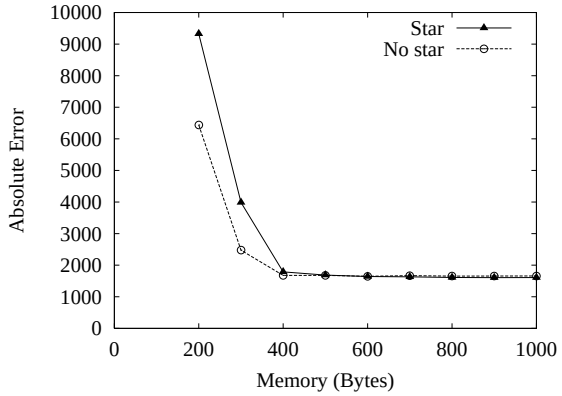

Figure 11: Memory and summarization (DBLP)

address the issue of how fast the tables "learn" the structure of the data by observing XPath queries and their result sizes.

To study convergence, we group the queries in our workloads into batches of 100 queries each, and we compute the average absolute error for each batch. Figures 7–9 show these errors for all three data sets and the four different query workloads corresponding to four different condition probabilities. The figures show that on-line annotated path tables have good convergence properties for all data sets and workloads.

6.3 Memory Requirement and Summarization

In this section, we investigate the effect of the amount of available memory on the accuracy of on-line annotated path tables. We also study the effectiveness of star paths in table summarization.

Figures 10–12 show the average estimation errors using on-line annotated path tables for the three data sets and different memory allocations. The figures show the errors for workloads with $p = 25\%$ and two methods of table summarization: using star path expressions, and *not* using star path expressions (i.e., discarding the table entries with low s_i values and losing the information they contain). The errors shown are for the last 800 queries in each query workload (the validation queries). The x-axis in each of these figures shows the target threshold, t_1. The trigger threshold, t_2, is always set to $2t_1$.

The figures show that giving on-line annotated path tables more memory results in an increase in estimation accuracy. However, the figures also show that estimation accuracy does not increase significantly when increasing the available memory beyond 700 bytes. The information captured by on-line annotated path tables does not require a lot of memory to represent, so the memory required for maximum accuracy will typically be in the range of hundreds of bytes. Such a small memory requirement is important if we have to build statistics for thousands of hidden Web data sources, as we expect the case will be if we want to query the entire Internet.

As for using star path expressions for summarization, the picture is not as clear. On the one hand, star path expressions allow us to retain some of the information contained in entries deleted from the path table, although at a coarser granularity. These star path expressions may, therefore, lead to increased estimation accuracy. This is the case for the SWISS-PROT data set (Figure 10).

On the other hand, the information contained in the star path expressions may have come from deleted path table entries with widely varying s_i values. In this case, the star path expressions can be an inaccurate representation of the deleted path table entries and may actually lead to a *decrease* in estimation accuracy. This is the case for the DBLP and synthetic data sets (Figures 11 and 12).

Fortunately, the difference in estimation accuracy be-

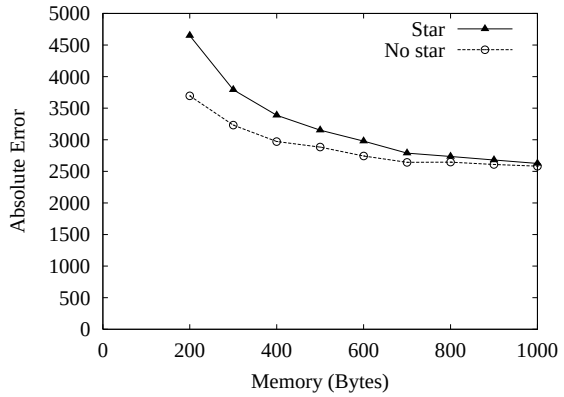

Figure 12: Memory and summarization (synthetic)

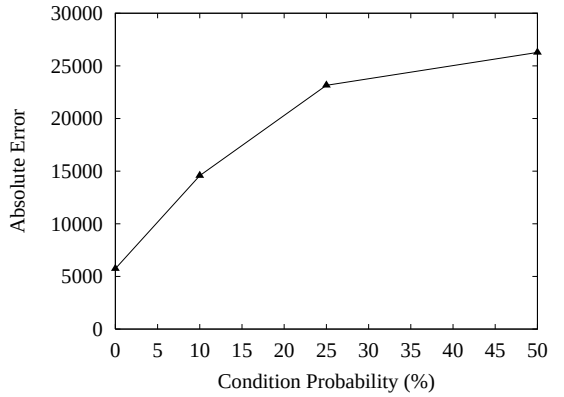

Figure 13: Conditions in the queries (SWISS-PROT)

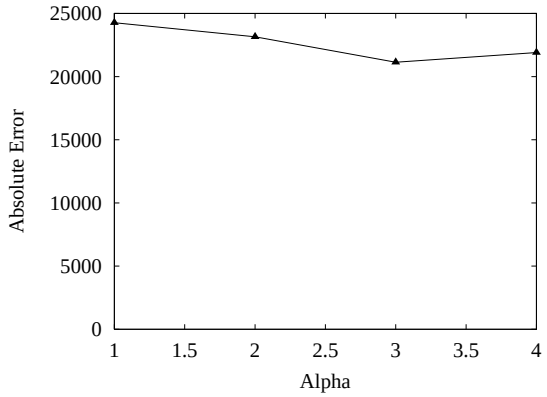

Figure 14: Varying α (SWISS-PROT)

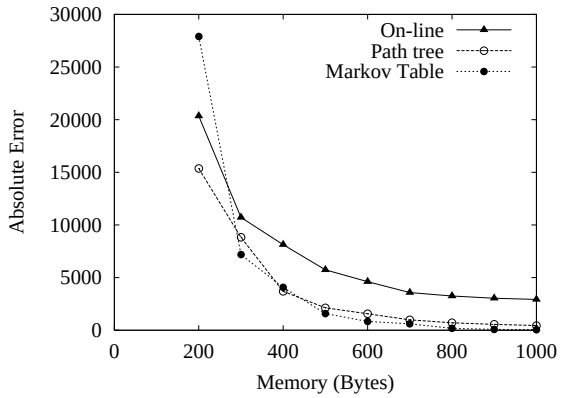

Figure 15: Comparison to static statistics (SWISS-PROT)

tween using and not using star path expressions is always small. Thus, the decision of whether or not to use these path expressions will have a minimal effect. We choose to be aggressive about retaining information about entries deleted from the on-line annotated path table during summarization, so we use star path expressions.

6.4 Effect of Conditions in the XPath Queries

On-line annotated path tables rely on the assumption that conditions have a uniform effect on selectivity. In this section, we investigate the effect of this simplifying assumption on estimation accuracy.

Figure 13 shows the average selectivity estimation error for the validation queries in workloads with different values of the condition probability parameter, p, on the SWISS-PROT data set. As p increases, the number of conditions in the query path expressions in the workload increases, so the assumption that conditions have a uniform effect on selectivity holds less and less. This leads to an increase in selectivity estimation error with increasing p, even as the average result size of the queries in the workload decreases with increasing p. However, the error remains adequately low. At its maximum ($p = 50\%$), the estimation error is around 20% of the average result size.

6.5 Sensitivity to the Parameter α

In this paper, we set the trigger threshold, t_2, using the formula $t_2 = \alpha t_1$. Figure 14 shows the estimation error for different values of the parameter α on the SWISS-PROT data set. As expected, increasing α leads to an increase in estimation accuracy, because it allows the on-line annotated path table to grow more and collect more information before triggering summarization, and because it reduces the number of table summarizations. However, the error is flat for all values of α. On-line annotated path tables are not sensitive to this parameter.

6.6 Comparison to Static XML Statistics

In this section, we compare the on-line XML statistics we propose in this paper to the static XML statistics that we proposed in [AAN01]. 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 [AAN01] 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 navigations 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* ($p = 0\%$).

Figures 15–17 show the selectivity estimation errors for workloads with no conditions using path trees and Markov tables as well as on-line annotated path tables. The errors are shown for the last 800 queries in the workloads (the

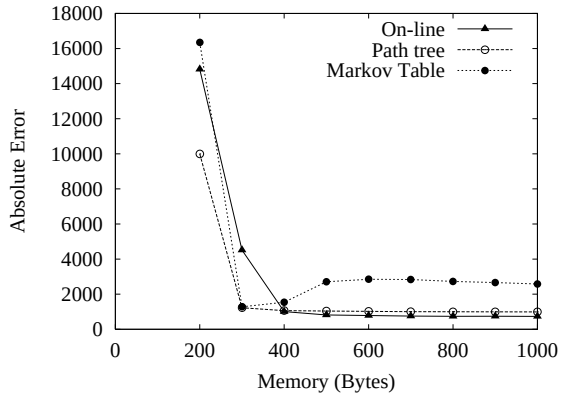

Figure 16: Comparison to static statistics (DBLP)

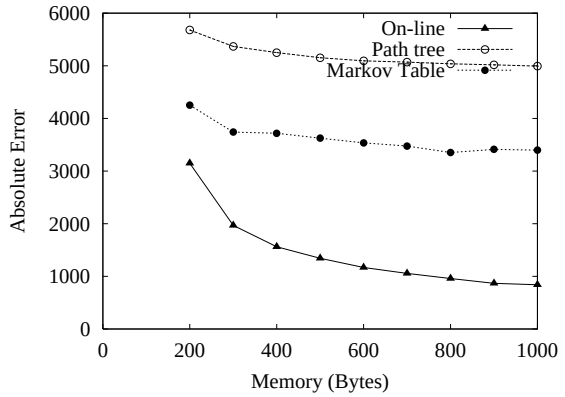

Figure 17: Comparison to static statistics (synthetic)

validation queries) for all three data sets and 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 figures show that on-line XML statistics are comparable in performance to static XML statistics, and sometimes even better.

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 15 and in small memory allocations in Figure 16. 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 16, and more strikingly in Figure 17, 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 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.

7 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 experimentally demonstrate using real and synthetic data sets that on-line annotated path tables have good convergence behavior, and that they work well across a wide range of parameter values. We also show that they are comparable to static XML statistics, and sometimes even better.

To the best of our knowledge, this paper is the first to address the issue of XML statistics for the hidden Web. Our goal in this work was to take a first step toward techniques that solve the daunting problem of gathering and using statistics for queries over the hidden web. As such a first step, our work opens a wide range of interesting possibilities 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.

In this paper, we assume that a condition on a tag in an XPath query has the same effect on selectivity as any other condition on this tag. Developing more elaborate techniques for handling XPath conditions is a possible area for future work.

Also, we do not try to infer any information about the structure of the XML tree from the results of the XPath queries. We only use selectivity information at the granularity of *whole path expressions*. A possible area for future work is inferring information about the structure of the XML tree based on the queries in the workload and their results. This may involve using heuristics, and it may also involve examining the results of the user queries in detail,

and not relying only on the sizes of these results as we do in this paper.

Finally, it may be possible to utilize semantic knowledge or schema knowledge to construct or refine statistics for hidden Web data sources.

References

[AAN01] Ashraf Aboulnaga, Alaa R. Alameldeen, and Jeffrey 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, September 2001.

[ACPS96] Sibel Adali, K. Selçuk Candan, Yannis Papakonstantinou, and V.S. 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.

[ANZ01] Ashraf Aboulnaga, Jeffrey F. Naughton, and Chun Zhang. Generating synthetic complex-structured XML data. In *Proc. 4th Int. Workshop on the Web and Databases (WebDB'2001)*, pages 79–84, Santa Barbara, California, May 2001.

[Bright] The deep web: Surfacing hidden value. White paper available from http://www.brightplanet.com/.

[CD99] James Clark and Steve DeRose (eds.). XML path language (XPath) version 1.0. W3C Recommendation available at http://www.w3.org/TR/xpath/, November 1999.

[Census] U.S. Census Bureau FactFinder database. http://factfinder.census.gov/.

[CFR+01] Don Chamberlin, Daniela Florescu, Jonathan Robie, Jérôme Siméon, and Mugur Stefanescu (eds.). XQuery: A query language for XML. W3C Working Draft available at http://www.w3.org/TR/xquery/, February 2001.

[CJK+01] Zhiyuan Chen, H.V. Jagadish, Flip Korn, Nick Koudas, S. Muthukrishnan, Raymond Ng, and Divesh Srivastava. Counting twig matches in a tree. In *Proc. IEEE Int. Conf. on Data Engineering*, pages 595–604, Heidelberg, Germany, April 2001.

[Clark99] James Clark (ed.). XSL transformations (XSLT) version 1.0. W3C Recommendation available at http://www.w3.org/TR/xslt/, November 1999.

[DBLP] DBLP Bibliography. http://www.informatik.uni-trier.de/~ley/db/.

[EDGAR] SEC EDGAR database. http://www.sec.gov/edgar.shtml.

[FLM98] Daniela Florescu, Alon Levy, and Alberto Mendelzon. Database techniques for the world-wide web: A survey. *SIGMOD Record*, 27(3):59–74, September 1998.

[IFF+99] Zachary G. Ives, Daniela Florescu, Marc Friedman, Alon Y. Levy, and Daniel 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.

[NDM+01] Jeffrey Naughton, David DeWitt, David Maier, et al. The Niagara Internet query system. *IEEE Data Engineering Bulletin*, 24(2):27–33, June 2001.

[NGT98] Hubert Naacke, Georges Gardarin, and Anthony Tomasic. Leveraging mediator cost models with heterogeneous data sources. In *Proc. IEEE Int. Conf. on Data Engineering*, pages 351–360, Orlando, Florida, February 1998.

[PGMW95] Yannis Papakonstantinou, Hector Garcia-Molina, and Jennifer Widom. Object exchange across heterogeneous information sources. In *Proc. IEEE Int. Conf. on Data Engineering*, pages 251–260, Taipei, Taiwan, March 1995.

[RGM01] Sriram Raghavan and Hector Garcia-Molina. Crawling the hidden web. In *Proc. Int. Conf. on Very Large Data Bases*, pages 129–138, Rome, Italy, September 2001.

[ROH99] Mary Tork Roth, Fatma Ozcan, and Laura 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, September 1999.

[SPROT] SWISS-PROT protein sequence database. http://www.expasy.ch/sprot/.

[Xalan] Xalan-C++. http://xml.apache.org/xalan-c/.

[Xyl01] Lucie Xyleme. A dynamic warehouse for XML data of the web. *IEEE Data Engineering Bulletin*, 24(2):40–47, June 2001.

[Zipf49] George K. Zipf. *Human Behaviour and the Principle of Least Effort.* Addison-Wesley, Reading, Massachusetts, 1949.

A Synthetic Data Set Used in Experiments

In our experiments, we use a synthetic data set generated using the XML data generator described in [ANZ01]. The tree representing the structure of this data set has 8 levels, with a total of 8643 nodes. The average fan outs of the 7 internal levels of this tree, from the root down, are 8, 5.2, 4.1, 4.6, 2.4, 2.5, and 2.5. The tree has 2161 nodes with repeated tag names.

The nodes of the tree have a Zipfian frequency distribution with parameter $z = 1$ [Zipf49]. The Zipfian frequencies are assigned to the tree nodes in *breadth first order*, with the root being assigned the lowest frequency and the rightmost leaf being assigned the highest frequency. The total frequency of all tree nodes, which is the total number of XML elements generated, is 1,000,000.

The spread of the number of child XML elements generated within a parent element is 75% around the mean number of such child XML elements.

The XML elements contain text words that follow a Zipfian distribution with parameter $z = 1$. We generate 10,000 distinct text words, and 1,000,000 total text words. All leaf XML elements have text words, and 25% of the internal XML elements have text words.