Estimating the Selectivity of XML Path Expressions for Internet Scale Applications

Ashraf Aboulnaga Alaa R. Alameldeen Jeffrey F. Naughton

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

Abstract

Data on the Internet is increasingly presented in XML format. This enables novel applications that pose queries over "all the XML data on the Internet." Queries over XML data use path expressions to navigate through the structure of the data, and optimizing these queries requires estimating the selectivity of these path expressions. In this paper, we propose two techniques for estimating the selectivity of simple XML path expressions over complex large-scale XML data as would be handled by Internet-scale applications: path trees and Markov tables. Both techniques work by summarizing the structure of the XML data in a small amount of memory and using this summary for selectivity estimation. We experimentally demonstrate the accuracy of our proposed techniques, and explore the different situations that would favor one technique over the other. We also demonstrate that our proposed techniques are more accurate than the best previously known alternative.

1 Introduction

Data on the Internet is increasingly presented in the *extensible markup language (XML)* format. The standardized, simple, self-describing nature of this format opens the door for novel Internet-scale applications that integrate and query XML data from numerous sources all over the Internet.

An example of such an Internet-scale application is the Niagara Internet query system [NDM+01]. Niagara allows a user to pose queries against "all the XML documents on the Internet," using an integrated search engine to find XML documents that are relevant to any query based on the path expressions that appear in it. Another Internet-scale query processor is Xyleme [Xyl], which aims to build an indexed, queryable XML repository of all the information on the World Wide Web.

**Proceedings of the 27th VLDB Conference,
Roma, Italy, 2001**

As an example of the queries that can be handled by an Internet-scale query processor, consider the following query expressed in the XQuery language [CFR+01]:

```
FOR $n_au IN document("*")//novel/author
    $p_au IN document("*")//play/author
WHERE $n_au/text()=$p_au/text()
RETURN $n_au
```

This is a join query that asks for writers who have written both a play and a novel. `document("*")` means that the query should be executed against all the known XML data on the Internet. In Niagara, the search engine would find all XML documents relevant to this query by finding all documents that contain the path `novel/author` or `play/author`.

Optimizing a query like this one requires estimating the selectivities of the path expressions it contains. For example, the query optimizer may need to estimate the selectivities of the path expressions `novel/author` and `play/author` (i.e., the number of `author` elements reachable by each path) to choose the more selective path expression as the outer data stream of the join. Path expressions are essential to querying XML, so estimating the selectivity of these path expressions is essential to XML query optimization.

Estimating the selectivity of XML path expressions requires having database statistics that contain information about the structure of the XML data. These statistics must fit in a small amount of memory because the query optimizer may consult them many times in the course of optimizing a query. The goal here is not to conserve memory, but rather to conserve *query optimization time*. The statistics must be small enough to be processed efficiently in the short time available for query optimization.

Ensuring that the statistics do not consume too much memory is particularly important for the Internet-scale applications that we focus on in this paper. It may be safe to assume that the structure of a single typical XML document can be captured in a small amount of memory. However, when considering Internet-scale applications that handle large amounts of XML data with widely varying structure, we cannot assume that the overall structure of all the XML data handled can be captured in a small amount of memory. The statistics used for selectivity estimation therefore have to be *summarized* so that they fit in the available memory.

In this paper, we present techniques for building database statistics that capture the structure of complex

XML data in a small amount of memory and for using these statistics to estimate the selectivity of XML path expressions. The focus of this paper is *simple path expressions*.

A simple path expression is a sequence of *tags* that represents a navigation through the tree structure of the XML data starting anywhere in the tree (not necessarily at the root). In abbreviated XPath syntax [CD99], which we use throughout the paper, a simple path expression of length n is expressed as $//t_1/t_2/\cdots/t_n$. This path expression specifies finding a tag t_1 anywhere in the document, and nested in it finding a tag t_2, and so on until we find a tag t_n. In this paper, we try to estimate the number of t_n elements reached by this navigation. Note that we assume an unordered model of XML and that we do not consider navigations based on IDREF attributes or on predicates on the attribute values.

We propose two techniques for capturing the structure of XML data for estimating the selectivity of path expressions. The first technique is to construct a tree representing the structure of the XML data, which we call the *path tree*. We then summarize this tree to ensure that it fits in the available memory by deleting low-frequency nodes and replacing them with nodes representing the information contained in the deleted nodes at a coarser granularity. The second technique is to store all paths in the data up to a certain length and their frequency of occurrence in a table of paths that we call the *Markov table*. We summarize the Markov table by deleting low-frequency paths, and we combine the paths of limited length in the Markov table to estimate the selectivity of longer paths.

The rest of this paper is organized as follows. Section 2 presents an overview of related work. Section 3 describes path trees. Section 4 describes Markov tables. Section 5 presents an experimental evaluation of the proposed techniques. Section 6 contains concluding remarks.

2 Related Work

Estimating the selectivity of XML path expressions is related to estimating the selectivity of substring predicates, which has been addressed in several papers [KVI96, JNS99, JKNS99, CKKM00]. These papers all use variants of the *pruned suffix tree* data structure. A suffix tree is a trie that stores all the strings in a database and all their suffixes. A pruned suffix tree is a suffix tree in which nodes corresponding to low frequency strings are pruned so that the tree fits in the available memory.

The techniques in [JNS99] and [CKKM00] are the basis for techniques developed in [CJK$^+$01] for estimating the selectivity of *twig queries*. Like path expressions, twig queries specify a navigation through the structure of XML documents or other tree-structured data. Twig queries are more general than the simple path expressions we consider in this paper. They can specify navigations based on *branching* path expressions, and they can specify specific *data values* that must be found at the ends of the path expressions (rather than navigating based only on the structure of the XML data).

The techniques developed in [CJK$^+$01] are the best previously known techniques that can be applied to the problem of estimating the selectivity of XML path expressions, even though they solve a more general problem. We restricted the data structures developed in [CJK$^+$01] for twig queries to the simpler problem of estimating the selectivity of path expressions by storing the minimum amount of information needed for selectivity estimation and no information about data values or path branching. We found that our data structures were able to give more accurate selectivity estimates for this simpler but very common case. The techniques developed in [CJK$^+$01] are described in more detail in Section 5.1.

Estimating the selectivity of XML path expressions requires summarizing the structure of the XML data. This can be done using *DataGuides* [GW97] or *T-indexes* [MS99]. These methods construct graphs that represent structural summaries of the data. For tree-structured XML data, the graphs constructed by both methods are identical, and they have the same structure as our unsummarized path trees. However, the problem of summarizing these graphs if they do not fit in the available memory is not addressed in [GW97] or [MS99]. On the other hand, the techniques we develop in this paper can summarize path trees to fit in any amount of memory.

Summarizing DataGuides is addressed in [GW99]. In that paper, a DataGuide is summarized by finding common labels in the paths represented in it or similar sets of objects reachable by these paths. This summary is not suitable for selectivity estimation because the frequency of occurrence of the paths does not play a role in summarization. Also, no summarization is possible if all the labels are distinct, and no guarantees can be made on the size of the summarized data guide.

The query optimizer of the Lore semi-structured database system estimates the selectivity of XML path expressions by storing selectivity information for all paths in the database of length up to k, where k is a tuning parameter [MW99]. This approach is valid but not scalable because the memory required for storing all paths of length up to k grows as the database grows. The Markov table approach that we propose in this paper also builds a table of all paths in the database up to a certain length, but this table is summarized if it overgrows the available memory. We also provide a method of combining the paths of limited length stored in the Markov table to obtain accurate selectivity estimates for longer path expressions.

Path expressions are used in object-oriented databases. Some cost models for query optimization in object-oriented databases depend on the selectivity of path expressions, but no methods for accurately estimating this selectivity have been proposed. See, for example, [GGT96].

3 Path Trees

In this section, we describe *path trees* that represent the structure of XML data and present techniques for summarizing these trees. We also describe using summarized path trees for selectivity estimation.

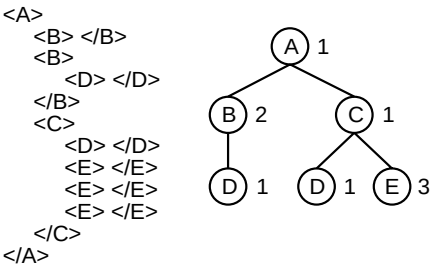


Figure 1: An XML document and its path tree

A path tree is a tree representing the structure of an XML document. Every node in the path tree represents a path starting from the root of the XML document. The root node of the path tree represents the root element of the document. A path tree node has a child node for every distinct tag name of an XML element directly nested in any of the elements reachable by the path it represents. Every path tree node is labeled with the *tag name* of the elements reachable by the path it represents and with the number of such elements, which we call the *frequency* of the node. Figure 1 presents an XML document and its path tree.

The path tree of an XML document can be constructed in one scan of the document using an event-based XML parser [SAX] and a stack that grows to the maximum nesting depth of the XML elements in the document. To construct a path tree for multiple XML documents, we create an artificial root node for all the XML data so that we can view it as a single tree.

A path tree contains all the information required for selectivity estimation. To estimate the selectivity of a query path expression, we scan the tree looking for all nodes with tags that match the first tag of the path expression. From every such node, we try to navigate down the tree following child pointers and matching tags in the path expression with tags in the path tree. This will lead us to a set of path tree nodes which all correspond to the query path expression. The selectivity of the query path expression is the total frequency of these nodes. This algorithm is $O(n)$, where n is the number of nodes in the path tree.

The problem with a path tree is that it may be larger than the available memory, so we need to summarize it. To summarize a path tree, we delete the nodes with the lowest frequencies from anywhere in the tree. We try to preserve some of the information represented in the deleted nodes at a coarser granularity by adding nodes to the path tree that represent groups of deleted nodes. These nodes that we add have the special tag name "$*$", which stands for "any tag name", so we call them $*$*-nodes* ("star nodes"). Next, we present four methods of summarizing path trees that differ in the amount of information they try to preserve in the $*$-nodes.

3.1 Sibling-$*$

In the first method for summarizing path trees, which we call *sibling-*$*$, we repeatedly choose the path tree node with the lowest frequency and mark it for deletion[1]. Marking a node for deletion does not reduce the size of the path tree. However, when we mark a node, A, for deletion, we check its siblings to see if they contain a node, B, that is either a $*$-node or a regular node that has been marked for deletion. If we find such a node, nodes A and B are *coalesced* into one $*$-node, which reduces the size of the path tree.

Each $*$-node represents multiple sibling nodes deleted from the path tree. The parent of a $*$-node is the parent of the deleted nodes it represents, and the children of these deleted nodes become children of the $*$-node. When a node, A, is coalesced with a $*$-node, the children of A become children of the $*$-node. Some of the children of A may have the same tag name as children of the $*$-node. Since these children are now siblings, the children with common tag names are coalesced, further reducing the size of the path tree. Coalescing the children of coalesced nodes is repeated recursively if needed.

Since both $*$-nodes and nodes with regular tags may be coalesced during summarization, all path tree nodes store the number of nodes in the original unsummarized path tree that they represent and the total frequency of these nodes. $*$-nodes always represent multiple nodes in the original path tree that have been deleted, while nodes with regular tags can represent either single nodes in the original path tree or multiple nodes with the same tag name that have been coalesced because their parents were coalesced.

During path tree summarization, we do not consider $*$-nodes as candidates for deletion. Coalesced nodes with regular tag names are deleted only if their *total* frequency is the lowest frequency in the path tree.

When the size of the path tree is reduced enough so that it fits in the available memory, we traverse the tree and compute for every $*$-node the average frequency of the multiple deleted nodes that it represents. This is the frequency that is used for selectivity estimation.

Nodes with regular tag names can represent multiple nodes in the original path tree if they are coalesced when their parents are coalesced. For such nodes, we do *not* always use the average frequency for selectivity estimation. We use the average frequency in some cases, but we use the total frequency in other cases. Thus, both the total frequency and the number of nodes represented have to be retained so that the average frequency can be computed when needed. Details are presented in the next section.

Figures 2 and 3 present a path tree of 12 nodes and its sibling-$*$ summarization to 9 nodes. The nodes of this tree are marked for deletion in the order A, I, J, E, H, D, C, G. Sibling nodes that can be coalesced are identified when marking J, D, and G. Coalescing G and H allows us to coalesce the two K nodes, saving us an extra node. The summarized path tree retains both the total frequency of the K node and the number of nodes it represents.

The $*$-nodes in sibling-$*$ summarization try to preserve the exact position of the deleted nodes in the original path tree. The cost of preserving this exact information is that we may need to delete up to $2n$ nodes to reduce the size of the tree by n nodes.

[1] This can be done in $O(n \log n)$ using a priority queue.

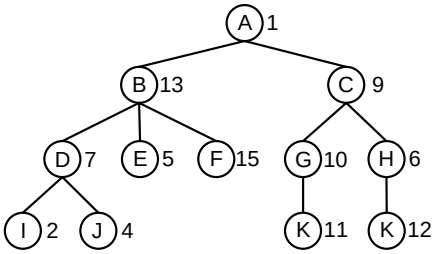


Figure 2: An example path tree

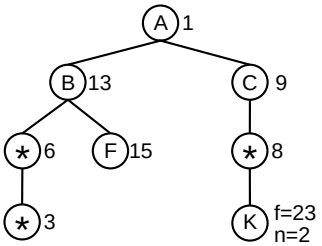


Figure 3: The sibling-$*$ summarization of Figure 2

3.2 Selectivity Estimation

To estimate the selectivity of a query path expression using a summarized path tree, we try to match the tags in the path expression with tags in the path tree to find all path tree nodes to which the path expression leads. The estimated selectivity is the total frequency of all these nodes. When we cannot match a tag in the path expression to a path tree node with a regular tag, we try to match it to a $*$-node that can take its place.

Tags in any position of the query path expression can be matched to $*$-nodes. For example, the path expression //A/B/C would match all of //A/$*$/C, //A/$*$/$*$, and //$*$/B/$*$. To find all the matches for a query path expression, we traverse the path tree looking for nodes whose tags match tags in any position of the path expression and start navigating from these nodes, matching with $*$-nodes when necessary. We allow matches with any number of $*$-nodes as long as they include at least one node with a regular tag name. We do not allow matches consisting entirely of $*$-nodes because there is not enough confidence in such a match.

When we match a tag from the query path expression with a $*$-node, we are making the assumption that this tag was present in the original path tree but was deleted and replaced with the $*$-node. We are essentially assuming that all query path expressions ask for paths that exist in the data, so we aggressively try to match them in the summarized path tree.

If a match of the query path expression in the path tree ends at a node, A, with a regular tag that represents multiple coalesced nodes of the original path tree, we check whether or not this match took us through a $*$-node. If the match took us through one or more $*$-nodes, node A contributes the *average* frequency of the nodes it represents to the estimated selectivity. A $*$-node encountered during the match represents multiple deleted nodes from the original path tree. We assume that if we were using the original unsummarized path tree, the match would have taken us through only one of the nodes represented by the $*$-node and ended at only one of the nodes represented by node A, so node A contributes its average frequency. On the other hand, if the match in the summarized path tree did not take us through a $*$-node, then the match in the unsummarized path tree would have taken us to *all* the nodes represented by node A, so node A contributes the *total* frequency of the nodes it represents to the estimated selectivity. This explains why nodes with regular tags that represent multiple coalesced nodes of the original path tree need to retain both the total frequency and the number of nodes they represent.

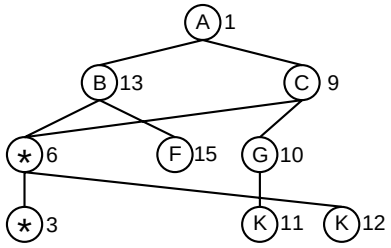


Figure 4: The level-$*$ summarization of Figure 2

3.3 Level-$*$

The second method for summarizing path trees, which we call *level*-$*$, has a $*$-node for every level of the path tree representing all deleted nodes at this level. As before, we delete the lowest frequency path tree nodes. All nodes deleted at any given level of the path tree are coalesced into the $*$-node for this level. The parents of these nodes become parents of the $*$-node and the children of these nodes become children of the $*$-node. This means that the path tree can become a *dag*. However, we can still us the same selectivity estimation algorithm as for sibling-$*$. As in sibling-$*$, when the children of a deleted node are added to the children of the corresponding $*$-node, any nodes that become siblings that have the same tag must be coalesced. Figure 4 shows the level-$*$ summarization of the path tree in Figure 2 to 9 nodes.

The $*$-nodes in level-$*$ summarization preserve only the level in the path tree of the deleted nodes, not their exact position as in sibling-$*$. Hence, level-$*$ usually deletes fewer nodes than sibling-$*$ to reduce the size of the tree by the same amount. To reduce the size of a path tree by n nodes, level-$*$ needs to delete fewer than $n + l$ nodes, where l is the number of levels in the tree.

3.4 Global-$*$

The third method for summarizing path trees is the *global*-$*$ method, in which a single $*$-node represents all low-frequency nodes deleted from anywhere in the path tree. The parents of the deleted nodes become parents of the $*$-node and their children become children of the $*$-node, so the path tree can become a *cyclic graph* with cycles involving the global $*$-node. Nevertheless, we can still use the same selectivity estimation algorithm as for sibling-$*$ and level-$*$. Figure 5 shows the global-$*$ summarization of the path tree in Figure 2 to 9 nodes.

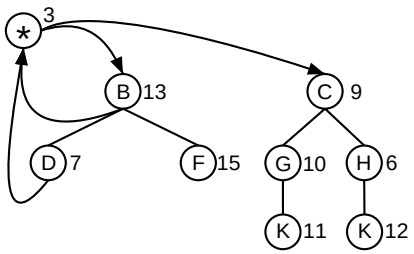


Figure 5: The global-$*$ summarization of Figure 2

Global-$*$ preserves less information about the deleted nodes than sibling-$*$ or level-$*$, so it has to delete fewer nodes. To reduce the size of a path tree by n nodes, global-$*$ deletes $n + 1$ nodes.

3.5 No-$*$

The final path tree summarization method, which we call *no-*$*$*, does not rely on $*$-nodes at all to represent deleted nodes. In the no-$*$ method, low-frequency nodes are simply deleted and not replaced with $*$-nodes. The path tree can become a *forest* with many roots. To reduce the size of a path tree by n nodes, no-$*$ deletes exactly n nodes. This is only one node less than global-$*$.

The fundamental difference between no-$*$ and the methods that use $*$-nodes is the effect of the absence of $*$-nodes on selectivity estimation. When matching a query path expression in a path tree summarized with no-$*$, if any tag in the path expression is not found, we assume that the entire path expression does not exist. No-$*$ conservatively assumes that nodes that do not exist in the summarized path tree did not exist in the original path tree. Methods that use $*$-nodes, on the other hand, aggressively assume whenever possible that nodes that do not exist in the summarized path tree *did* exist in the original path tree but were deleted and replaced with $*$-nodes. The characteristics of the query workload determine which of these two assumptions is more accurate.

4 Markov Tables

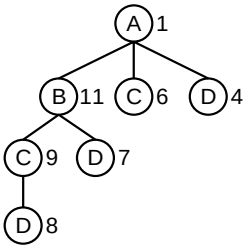
In this section, we describe a different method of representing the structure of XML data for selectivity estimation.

We construct a table of all the distinct paths in the data of length up to m and their frequency, where m is a parameter ≥ 2. The table provides selectivity estimates for all path expressions of length up to m. To estimate the selectivity of longer path expressions, we combine several paths of length m using the formula

$$f(t_1/t_2/\cdots/t_n) = f(t_1/t_2/\cdots/t_m) \times \prod_{i=1}^{n-m} \frac{f(t_{1+i}/t_{2+i}/\cdots/t_{m+i})}{f(t_{1+i}/t_{2+i}/\cdots/t_{m+i-1})} \quad (1)$$

where $f(t_1/t_2/\cdots/t_n)$ is the frequency of the path $t_1/t_2/\cdots/t_n$. $f(t_1/t_2/\cdots/t_k)$ for any $k \leq m$ is obtained by a lookup in the table of paths[2].

[2]This can be done in $O(1)$ by using a hash table.



Path	Freq	Path	Freq
A	1	AC	6
B	11	AD	4
C	15	BC	9
D	19	BD	7
AB	11	CD	8

Figure 6: A path tree and the corresponding Markov table

For example, if $m = 3$ and the query path expression is //A/B/C/D, the formula used would be:

$$f(\mathsf{A/B/C/D}) = f(\mathsf{A/B/C})\frac{f(\mathsf{B/C/D})}{f(\mathsf{B/C})}$$

The fraction $f(\mathsf{B/C/D})/f(\mathsf{B/C})$ can be interpreted as the average number of D elements contained in all B/C paths.

In this approach, we are assuming that a tag in any path in the XML data depends only on the $m-1$ tags preceding it. We are, in effect, modeling the paths in the XML data as a Markov process of order $m-1$, so we call the table of paths that we use the *Markov table*. The "short memory" assumption made in this approach is very intuitive, and we expect it to hold for most XML data, even for $m = 2$ or 3. This assumption is also used for selectivity estimation in [JNS99] and [CJK$^+$01]. Figure 6 presents a path tree and its corresponding Markov table for $m = 2$.

Markov tables represent an accurate approximation of the structure of the XML data based on the short memory assumption, but they may not fit in the available memory. As we did for path trees, we summarize Markov tables by deleting low-frequency paths.

Low-frequency paths of length 1 or 2 that are deleted from the Markov table are replaced with special $*$-*paths* ("star paths") that preserve some of the information lost by deletion. These $*$-paths are very similar to the $*$-nodes used in path tree summarization. Low-frequency paths of length greater than 2 (for $m > 2$) are discarded and not replaced with $*$-paths. If estimating the selectivity of a query path expression using Equation 1 involves looking up a path of length > 2 that is not found in the Markov table, we switch to using Equation 1 with paths of length 1 and 2 (i.e., with $m = 2$). This corresponds to using a Markov process of order 1.

Next, we describe three methods of summarizing Markov tables that differ in the way they use $*$-paths to handle deleted paths of length 1 and 2. For all these methods, deleted paths of length greater than 2 are discarded and not replaced with $*$-paths.

4.1 Suffix-$*$

The first method for summarizing Markov tables, which we call *suffix-*$*$*, has two special $*$-paths: a path, $*$, representing all deleted paths of length 1, and a path, $*/*$, representing all deleted paths of length 2. When deleting a low-frequency path of length 1, it is added to the path $*$.

When deleting a low-frequency path of length 2, we do *not* add it to the path $*/*$ right away.

We keep a set of deleted paths of length 2, S_D. When we delete a path of length 2, say A/B, we look for any path in the set S_D that starts with the same tag as the path being deleted, in this case A. If no such path is found, we remove the path being deleted, A/B, from the Markov table and add it to S_D. If we find such a path in S_D, say A/C, we remove A/B from the Markov table, remove A/C from S_D, and add a new path A/$*$ that represents these two paths to the Markov table.

A/$*$ represents all deleted paths that start with the tag A. We call the path A/$*$ a "suffix-$*$" path. When we delete a path of length 2, before we check the set S_D, we check the Markov table to see if there is a suffix-$*$ path that has the same starting tag as the path being deleted. If we find such a path, the path being deleted is combined with it. In our example, if we delete A/D, we would combine it with A/$*$.

Suffix-$*$ paths in the Markov table are considered for deletion based on the *total* frequency of the deleted paths that they represent. When a suffix-$*$ path is deleted, it is added to the path $*/*$.

In our example, paths A/B and A/C individually qualify for deletion because of their low frequency. Their total frequency when they are combined into A/$*$ may be high enough to prevent them from being deleted. If at some point during summarization the total frequency of A/$*$ is the lowest frequency in the Markov table, it is deleted and added to the path $*/*$.

This summarization algorithm is a greedy algorithm that may miss opportunities for combining paths. However, it is simple and practical, and it achieves good results.

At the end of summarization, paths still remaining in S_D are added to the path $*/*$, and the average frequencies of all $*$-paths are computed. When selectivity estimation uses a $*$-path, it uses the average frequency of all deleted paths represented by this $*$-path.

To estimate the selectivity of a query path expression, we try to use Equation 1 with the maximum m in the Markov table. If any of the required paths is not found in the table, we switch to using Equation 1 with $m = 2$ for the entire path expression. In this case, if a required path of length 1 is not found, we use the frequency of the path $*$. If a required path of length 2, say A/B, is not found, we look for a path A/$*$. If we find it, we use its frequency. Otherwise we use the frequency of the path $*/*$. If all the paths used for estimation are $*$-paths, we estimate the selectivity of the query path expression to be zero, because we consider that there is not enough confidence in the result of Equation 1 in this case.

4.2 Global-$*$

The second method for summarizing Markov tables, which we call *global-$*$*, has only two $*$-paths: a path, $*$, representing all deleted paths of length 1, and a path, $*/*$, representing all deleted paths of length 2. When deleting a low-frequency path of length 1 or 2, it is immediately added to the appropriate $*$-path. Selectivity estimation is the same as for suffix-$*$. Global-$*$ does not preserve as much information about deleted paths as suffix-$*$, but it may delete fewer paths to summarize the Markov table.

4.3 No-$*$

The final method for summarizing Markov tables, which we call *no-$*$*, does not use $*$-paths. Low-frequency paths are simply discarded.

When using Equation 1 for selectivity estimation using a Markov table summarized with no-$*$, if any of the required paths for $m = 2$ is not found, we estimate a selectivity of zero. No-$*$ for Markov tables is similar to no-$*$ for path trees. It conservatively assumes that paths that do not exist in the summarized Markov table did not exist in the original Markov table.

5 Experimental Evaluation

In this section, we present an experimental evaluation of our proposed techniques using real and synthetic data sets. We determine the best summarization methods for path trees and Markov tables and the conditions under which each technique wins over the other. We also compare our proposed techniques to the best known alternative: the pruned suffix trees of [CJK$^+$01].

5.1 Estimation Using Pruned Suffix Trees

To estimate the selectivity of path expressions and the more general twig queries, [CJK$^+$01] proposes building a trie that represents all the path expressions in the data. For every root-to-leaf path in the data, the path and all its suffixes are stored in the trie. Every possible path expression in the data is therefore represented by a trie node. Every trie node contains the total number of times that the path it represents appears in the data. To prune the trie so that it fits in the available memory, the low-frequency nodes are deleted. To avoid deleting internal trie nodes without deleting their descendants, pruning is done based on the total frequency of the node and all its descendants.

To estimate the selectivity of a query path expression, the *maximal (i.e., longest) sub-paths* of this path expression that appear in the trie are determined, and their frequencies are combined in a way that is very similar to the way we combine frequencies in our Markov table technique.

In [CJK$^+$01], every node of the trie stores a hash signature of the set of nodes that the path expression it represents is rooted at. These hash signatures are used to combine the selectivities of multiple paths to estimate the selectivity of branching path expressions, or *twig queries*. Since we do not consider branching path expressions, we do not store hash signatures in the nodes of the trie that we use in our experiments. The trie with set hash signatures is referred to in [CJK$^+$01] as a *correlated suffix tree*. In this paper, we refer to the trie that does not include set hash signatures as the *pruned suffix tree*.

Note that Markov tables bear some similarity to pruned suffix trees. A key difference between these two techniques is that Markov tables only store paths of length up to m while pruned suffix trees store paths that may be of any length and that may contain tags that are not needed due to the "short memory" property. Furthermore, the summarization methods for Markov tables are very different from the summarization method for pruned suffix trees.

5.2 Data Sets

We present the results of experiments on one synthetic and one real data set. The synthetic data set has 1,000,000 XML elements. Its unsummarized path tree has 3197 nodes and 6 levels, requiring 38KB. The unsummarized Markov tables for $m = 2$ and 3 require 60KB and 110KB, respectively. The frequencies of the path tree nodes follow a Zipfian distribution with skew parameter $z = 1$. The Zipfian frequencies are assigned in ascending order to the path tree nodes in breadth first order (i.e., the root node has the lowest frequency and the rightmost leaf node has the highest frequency). 50% of the internal nodes of this path tree have repeated tag names, which introduces "Markovian memory" in the data. For example, if two internal nodes of the path tree have tag name A, and only one of these nodes has a child node B, then if we are at a node A, whether or not this node has a child B will depend on *which* A node this is, which in turn depends on how we got to this node from the root node. More details about our synthetic data generation process can be found in [ANZ01].

The real data set is the DBLP bibliography database, which has 1,399,765 XML elements. Its unsummarized path tree has 5883 nodes and 6 levels, requiring 69KB. The unsummarized Markov tables for $m = 2$ and 3 require 20KB and 98KB, respectively.

5.3 Query Workloads

We present results for two workloads for every data set. Each workload consists of 1000 query path expressions having a random number of tags between 1 and 4.

The query path expressions in the first workload, which we call the *random paths* workload, consist of paths that are chosen at random from the path tree of the data set. Thus, all queries have non-zero result sizes. This workload models a user who knows the structure of the data well, and so asks for paths that exist in the data.

The query path expressions in the second workload, which we call the *random tags* workload, consist of random concatenations of the tags that appear in the data set. In this workload, most query path expressions of length 2 or more have a result size of zero. This workload models a user who knows very little about the structure of the data.

The average result sizes of the random paths and random tags workloads on the synthetic data set are 491 and 71, respectively. For the DBLP data set, the average result sizes of the random paths and random tags workloads are 36,060 and 343, respectively.

5.4 Performance Evaluation Method

We present the performance of the different selectivity estimation techniques in terms of their *average absolute error* for all queries in the workload. The conclusions from the relative error are the same, but the relative error is not defined for many queries in the random tags workloads because their actual result size is 0.

For a given data set and query workload, we vary the available memory for the different selectivity estimation techniques from 5KB to 50KB and present the average absolute error for the 1000 queries in the workload for each technique at each memory setting.

In all the data structures used for estimation, tag names are not stored as character strings. Instead, we hash the tag names and store their hash values. This conserves memory because a tag requires one integer of storage regardless of its length.

5.5 Summarizing Path Trees

In this section, we illustrate the best summarization methods for path trees. Figures 7(a) and (b) present the average absolute error in selectivity estimation using path trees summarized in different ways for the random paths and random tags workloads on the synthetic data set, respectively.

Figure 7(a) shows that, for the random paths workload, all summarization methods that use $*$-nodes have similar performance, and they are all better than the no-$*$ method. The methods using $*$-nodes are better than no-$*$ because the query path expressions ask for paths that exist in the data, so the aggressive assumption that these methods make about nodes not in the summarized path tree are mostly valid and result in higher accuracy. Since all methods using $*$-nodes have similar performance, we conclude that the more detailed information maintained by the more complex sibling-$*$ and level-$*$ methods does not translate into higher estimation accuracy. Hence, for workloads that ask for paths that exist in the data, global-$*$ is the best path tree summarization method.

The situation is different for the random tags workload in Figure 7(b). Since query path expressions in the random tags workload ask for paths that mostly do not exist in the data, the correct thing to do when we are unable to match the entire query path expression with nodes in the path tree is to estimate a selectivity of zero. This is what no-$*$ does. The methods that use $*$-nodes are misleading in this case because they allow us to match tags in the query path expression with $*$-nodes in the path tree even when the query path expression does not exist in the data. This results in significantly less accuracy than no-$*$. Hence, for workloads that ask for paths that do not exist in the data, no-$*$ is the best path tree summarization method.

5.6 Summarizing Markov Tables

In this section, we illustrate the best summarization methods for Markov tables. For all data sets and query workloads, we observe that unsummarized Markov tables with $m = 3$ are very accurate, so we only evaluate the performance of summarized Markov tables with $m = 2$ and 3,

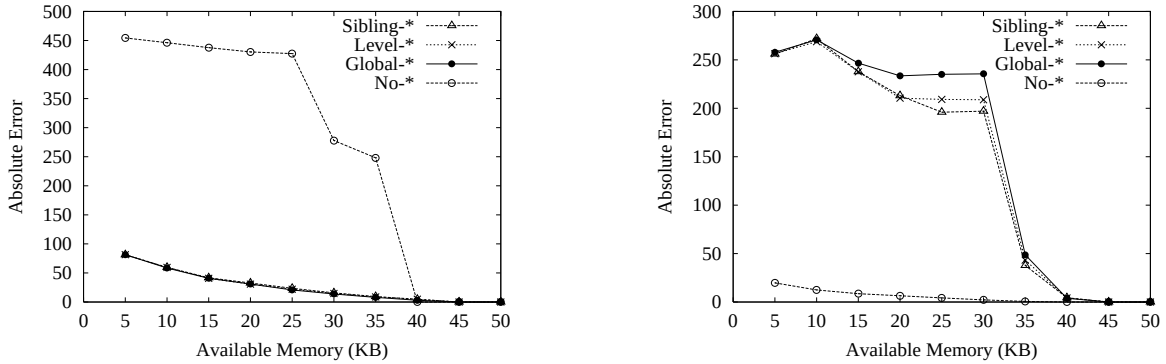


Figure 7: Path tree summarization, synthetic data set, (a) random paths and (b) random tags

but not with $m > 3$. In general, the practical values of m are 2 and 3.

Figures 8(a) and (b) present the estimation accuracy using Markov tables summarized in different ways for the random paths and random tags workloads on the synthetic data set, respectively.

Figure 8(a) shows that, for the random paths workload, suffix-$*$ summarization is best. Unlike for path trees, the summarization method that preserves the most information about deleted paths works best for Markov tables. $m = 2$ and $m = 3$ have similar performance, so the conclusion is to use the simpler $m = 2$. Thus, the best Markov table approach for workloads that ask for paths that exist in the data is to use $m = 2$ and suffix-$*$ summarization.

Figure 8(b) shows that, for the random tags workload, $m = 2$ and global-$*$ or no-$*$ summarization are the best methods. No-$*$ works well for the same reason that it works well in path tree summarization for the random tags workload. Global-$*$ is similar in performance to no-$*$ because many of the query path expressions that ask for paths that do not exist in the data get matched entirely with the paths $*$ and $*/*$, so they have an estimated selectivity of zero, which is correct. The best Markov table approach for workloads that ask for paths that do not exist in the data is, therefore, to use $m = 2$ and the simpler no-$*$ summarization.

5.7 Estimation Accuracy

In this section, we compare the best techniques for path trees and Markov tables as identified in the previous sections. We also compare these techniques to the pruned suffix tree approach.

Figure 9(a) presents the selectivity estimation errors for the random paths workload on the synthetic data set using path trees summarized with global-$*$, Markov tables with $m = 2$ summarized with suffix-$*$, and pruned suffix trees. Figure 9(b) presents the selectivity estimation errors for the random tags workload on the synthetic data set using path trees summarized with no-$*$, Markov tables with $m = 2$ summarized with no-$*$, and pruned suffix trees. Figures 10(a) and (b) present the same information for the DBLP data set.

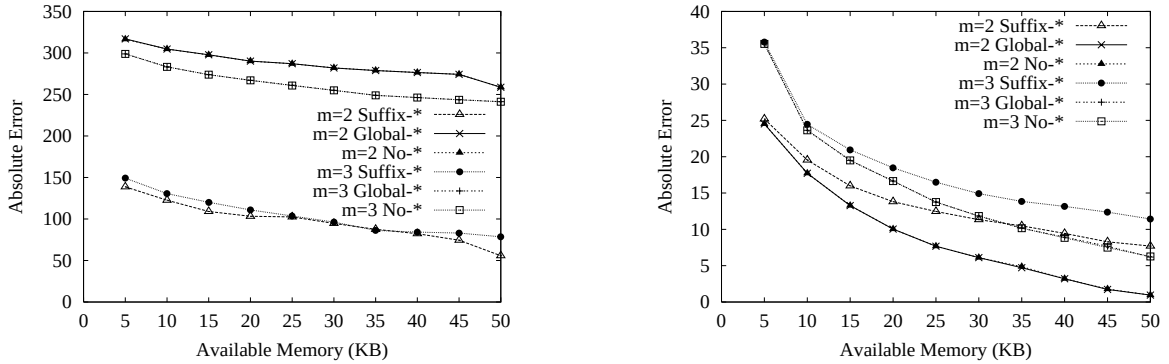
For the synthetic data set, path trees are the most accurate technique, and both path trees and Markov tables are more accurate than pruned suffix trees. For the DBLP data set, Markov tables are the most accurate technique, and they are much more accurate than pruned suffix trees. Path trees, on the other hand, are the least accurate technique for the DBLP data set. Their estimation error, which is too high to show in Figure 10, is usually greater than 100.

The DBLP data set represents bibliography information for many different conferences and journals. Each conference or journal is a different sub-tree of the path tree, but the structure of the data within each of these sub-trees is the same for any conference or journal. Path tree summarization cannot compactly represent the common structure of these sub-trees. On the other hand, Markov tables, and to a lesser extent pruned suffix trees, can effectively capture this common structure. For example, note that each bibliography entry in every conference or journal in the DBLP data set has one or more `author` elements. In the path tree, there will be an `author` node in the sub-tree corresponding to every conference or journal. Some of these nodes will have to be deleted during tree summarization resulting in a loss of accuracy. In the Markov table, on the other hand, there will only be one `author` path for all the `author` nodes in the path tree.

Thus, if the data has many common sub-structures, Markov tables should be used. If the data does not have many common sub-structures, path trees should be used. Choosing the appropriate selectivity estimation technique always results in higher accuracy than using pruned suffix trees.

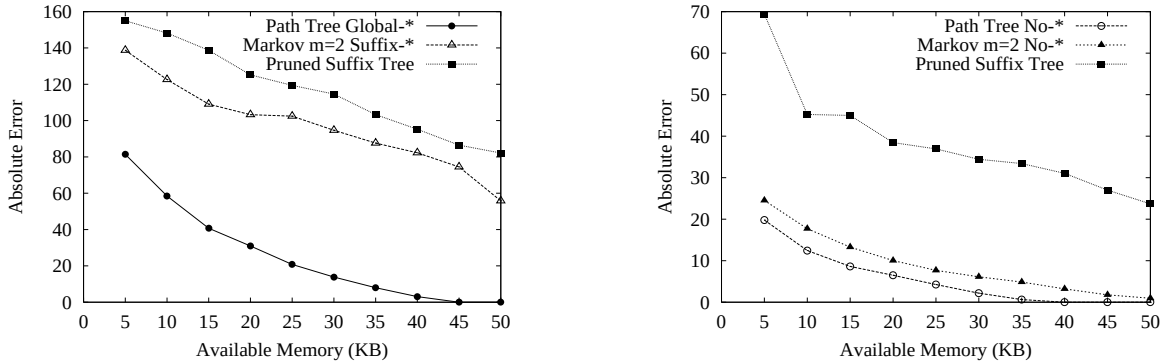
6 Conclusions

The proliferation of XML on the Internet will enable novel applications that query "all the data on the Internet." The queries posed by these applications will involve path expressions, and optimizing these queries will require estimating the selectivity of these path expressions. In this paper, we presented two techniques for summarizing the structure of large-scale XML data in a small amount of memory for estimating the selectivity of XML path expressions: path trees and Markov tables.



(a) (b)

Figure 8: Markov table summarization, synthetic data set, (a) random paths and (b) random tags



(a) (b)

Figure 9: Estimation accuracy, synthetic data set, (a) random paths and (b) random tags

The correct selectivity estimation technique to use depends on the XML data whose structure is being summarized. If the data has a lot of common structures, Markov tables with $m = 2$ should be used. If the data does not have such common structures, path trees should be used.

The best way to summarize path trees and Markov tables depends on the characteristics of the query workload. If the query path expressions ask for paths that exist in the data, then the aggressive global-$*$ and suffix-$*$ techniques should be used for summarizing path trees and Markov tables, respectively. If the query path expressions ask for paths that do not exist in the data, then the conservative no-$*$ technique should be used for both path trees and Markov tables.

The correct choice from our techniques always results in higher selectivity estimation accuracy than pruned suffix trees [CJK$^+$01], the best previously known alternative. The techniques in [CJK$^+$01] solve a more general problem, so their full power is not evident in the comparison with our proposed techniques that solve a simpler problem. It is still an open question whether XML query optimizers will require selectivity information about simple path expressions, in which case our proposed techniques would be better, or about more complex path expressions involving branches and values, in which case the techniques in [CJK$^+$01] would be better. Answering this question is a possible area for future work.

At this time, we cannot conclusively determine the typical characteristics of XML data that will be available on the Internet, so we cannot recommend an overall best technique for selectivity estimation. However, if we were to make an educated guess, we would say that things like standard DTDs and schema libraries will result in a lot of common sub-structures. We would also guess that users typically know enough about the semantics of the tag names to ask for paths that generally do exist in the data. Thus, we would recommend Markov tables with $m = 2$ and suffix-$*$ summarization. If this guess proves to be wrong, we simply need to choose another one of our proposed techniques. In any case, developing a general framework for choosing the correct selectivity estimation technique is an interesting topic for future work.

Acknowledgements

We thank Zhiyuan Chen and Divesh Srivastava for providing us with the code for pruned suffix trees and helping us with this code, and for providing us with a real XML data set that we used in some of our experiments. Ashraf Aboulnaga and Jeff Naughton were funded by NSF through grants CDA-9623632 and ITR 0086002, and by DARPA through NAVY/SPAWAR Contract No. N66001-99-1-8908. Alaa Alameldeen was funded by NSF through grant EIA-9971256.

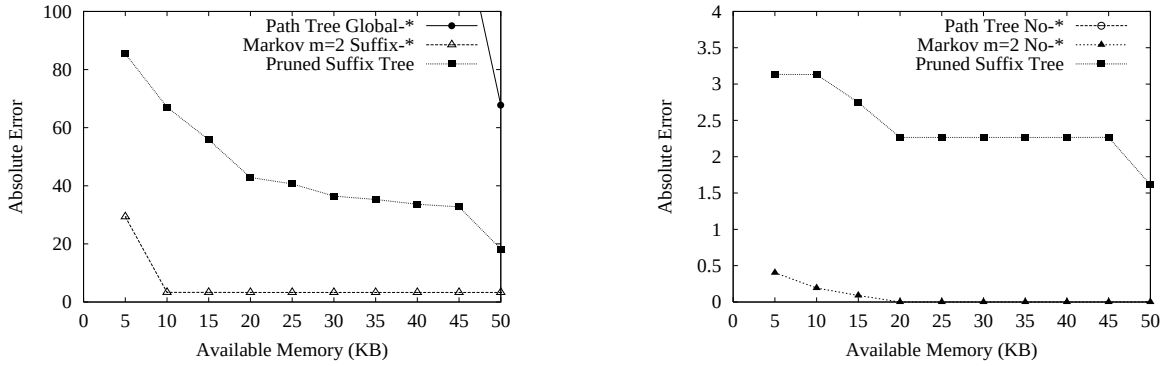


Figure 10: Estimation accuracy, DBLP data set, (a) random paths and (b) random tags

References

[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)*, Santa Barbara, California, May 2001.

[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.

[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.

[CKKM00] Zhiyuan Chen, Flip Korn, Nick Koudas, and S. Muthukrishnan. Selectivity estimation for boolean queries. In *Proc. ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 216–225, Dallas, Texas, May 2000.

[GGT96] Georges Gardarin, Jean-Robert Gruser, and Zhao-Hui Tang. Cost-based selection of path expression processing algorithms in object-oriented databases. In *Proc. Int. Conf. on Very Large Data Bases*, pages 390–401, Mumbai (Bombay), India, September 1996.

[GW97] Roy Goldman and Jennifer Widom. DataGuides: Enabling query formulation and optimization in semistructured databases. In *Proc. Int. Conf. on Very Large Data Bases*, pages 436–445, Athens, Greece, August 1997.

[GW99] Roy Goldman and Jennifer Widom. Approximate DataGuides. In *Proc. Workshop on Query Processing for Semistructured Data and Non-standard Data Formats*, Jerusalem, Israel, January 1999.

[JKNS99] H.V. Jagadish, Olga Kapitskaia, Raymond T. Ng, and Divesh Srivastava. Multi-dimensional substring selectivity estimation. In *Proc. Int. Conf. on Very Large Data Bases*, pages 387–398, Edinburgh, Scotland, September 1999.

[JNS99] H.V. Jagadish, Raymond T. Ng, and Divesh Srivastava. Substring selectivity estimation. In *Proc. ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 249–260, Philadelphia, Pennsylvania, May 1999.

[KVI96] P. Krishnan, Jeffrey Scott Vitter, and Bala Iyer. Estimating alphanumeric selectivity in the presence of wildcards. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 282–293, Montreal, Canada, June 1996.

[MS99] Tova Milo and Dan Suciu. Index structures for path expressions. In *Proc. 7th Int. Conf. on Database Theory*, pages 277–295, Jerusalem, Israel, January 1999.

[MW99] Jason McHugh and Jennifer Widom. Query optimization for XML. In *Proc. Int. Conf. on Very Large Data Bases*, pages 315–326, Edinburgh, Scotland, September 1999.

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

[SAX] SAX 2.0: The simple API for XML. http://www.megginson.com/SAX/index.html.

[Xyl] Xyleme home page. http://www.xyleme.com.