SF-Tree: An Efficient and Flexible Structure for Estimating Selectivity of Simple Path Expressions with Statistical Accuracy Guarantee

Wai-Shing Ho, Ben Kao, David W. Cheung, YIP Chi Lap [Beta], and Eric Lo

Department of Computer Science and Information Systems
The University of Hong Kong, Hong Kong
{wsho, kao, dcheung, clyip, ecllo}@csis.hku.hk

Abstract. Estimating the *selectivity* of a *simple path expression* (*SPE*) is essential for selecting the most efficient evaluation plans for XML queries. To estimate selectivity, we need an *efficient* and *flexible* structure to store a summary of the path expressions that are present in an XML document collection. In this paper we propose a new structure called *SF-Tree* to address the selectivity estimation problem. SF-Tree provides a flexible way for the users to choose among accuracy, space requirement and selectivity retrieval speed. It makes use of *signature files* to store the SPEs in a *tree* form to increase the selectivity retrieval speed and the accuracy of the retrieved selectivity. Our analysis shows that the probability that a selectivity estimation error occurs decreases *exponentially* with respect to the error size.

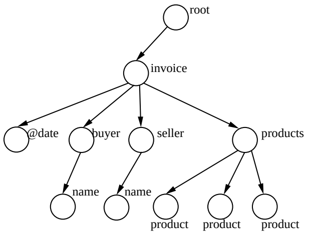
Keywords: SF-Tree, query processing, selectivity estimation, XML, path expressions.

1 Introduction

Extensible Markup Language (*XML*) [15] is becoming the standard of information exchange over the Internet. The standardized and self-describing properties of XML documents make them ideal for information exchange. An XML document can be regarded as a textual representation of a directed graph called *data graph* that consists of *information items* such as *elements* and *attributes*. Figure 1 shows an XML document representing an invoice and its data graph. Each node in the graph corresponds to an information item, and the edges represent the nesting relationships between the information items. Each information item has a *tag* (or *label*, *name*) that describes its semantics. If we ignore the `IDREF`s, a data graph is a tree.

```
<invoice date="1/7/2002">
   <buyer><name>ABC Corp.</name></buyer>
   <seller><name>D.com</name></seller>
   <products>
      <product>TV</product>
      <product>VCD player</product>
      <product>VCR</product>
   </products>
</invoice>
```

(a) An XML invoice



(b) The corresponding data graph

Fig. 1. An XML invoice and its data graph

```
FOR $i IN document("*")//supermarket/product
    $j IN document("*")//store/product
WHERE $i/text()=$j/text()
RETURN <diff> $i/price - $j/price </diff>
```

Fig. 2. An example query to an eMarket Web site

In addition to texts, structures of XML documents (i.e., how the information items nest each other) also carry information. To extract useful information out of XML documents, we need query languages such as *XQuery* [17] and *XPath* [16] that allow users to query the structures of XML documents. Figure 2 shows an example XQuery that queries an XML-based eMarket Web site that stores the information of the products sold in supermarkets or stores. It calculates the price differences of any products between a supermarket and a store. There are different ways (or *evaluation plans*) to evaluate the previous query. For example, we may first find out a list of all the products that are sold in supermarkets by using a path index, and then verify if those products are also sold in any stores by navigating the XML documents. Alternatively, we may first find out all the products that are sold in the stores. The only difference between the previous two evaluation plans is the order of evaluating the path expressions "`//supermarket/product`" and "`//store/product`". To increase the query processing efficiency, we should first evaluate the path expression that returns fewer elements. In this case, most of the irrelevant elements are filtered out by efficient path indices resulting in less amount of navigation, and hence better efficiency. Thus, in order to choose an efficient evaluation plan, we have to efficiently retrieve the *counts* (or, *selectivities*) of elements that are returned by any path expressions. Our focus in this paper is on estimating the selectivities of *simple path expressions* (*SPEs*). An SPE queries the structure of an XML document. It is denoted by a sequence of tags "$//t_1/t_2/\cdots/t_n$" that represents a navigation through the structure of an XML document. The navigation may start anywhere in the document but each transition goes from a parent to a child. An SPE returns all the information items t_n that can be reached by the navigation specified in it. The selectivity $\sigma(p)$ of an SPE p is the number of information items that the SPE returns.

1.1 Related Work

To get the selectivities of SPEs, we need some data structures to store the statistics of an XML document. There are different approaches to storing those statistics [1, 5, 9–12], but they all suffer from various drawbacks. Path indices like *path tree* [1], *1-index* [8], and *DataGuide* [5] are automata that can recognize all *absolute path expressions* (*APEs*) of an XML document. A path tree has a similar structure as a data graph except that siblings with the same name are coalesced into a node. Hence if we combine the three "`product`" nodes in Figure 1 into one, it becomes the path tree for our invoice. We can associate the selectivities of APEs to their corresponding states in the automaton. However, in these structures we need a full walk of the automaton to retrieve the selectivity of an SPE since an SPE can start anywhere. If the document is very complex or has an irregular structure (e.g., consider XML documents that are collected from various data sources), the automata is large and hence the full walk is inefficient. An efficient way to store the statistics is to use structures such as suffix tries [2] and hash tables [7] to store the association between SPEs and selectivity counts. However, suffix tries and hash tables require a large amount of space and they will be too large to be stored and accessed efficiently for complex documents.

Many researchers observe that exact selectivities are usually not required [1, 2, 9, 11, 12]. Thus they propose *approximate* structures to *estimate* the selectivities in order to increase the space and time efficiency. However, none of these approaches provide any *accuracy guarantees* to the retrieved selectivities.

1.2 Contributions

This paper proposes a new data structure called *SF-Tree* that efficiently stores the counts/selectivities of all SPEs in an XML document. In an SF-Tree SPEs are grouped by their selectivities. Thus the selectivity of an SPE can be found by finding to which group the SPE belongs. In order to increase the accuracy and efficiency of retrieving a selectivity, we use *signature files* to summarize the groups and organize them in form of a *tree*. We will show in this paper that using SF-Tree for estimating XML SPE selectivity is:

- *efficient*. We have a large speed up in selectivity estimation time over path trees. Our analysis shows that the selectivity estimation time is *independent* of data size. Thus SF-Tree is especially efficient compared with other structures such as path tree and suffix trie when the data is complex.
- *accurate*. Although SF-Tree does not always return exact selectivities, we have a tight *statistical accuracy guarantee* on the retrieved selectivities. This guarantee, which bounds the probability for an SF-Tree to report an incorrect selectivity, makes SF-Tree superior to all previous approaches that use approximate structures to estimate selectivities. We will show in our analysis and experiments that SF-Tree returns exact selectivity in most cases.
- *flexible*. By using different parameters, SF-Tree allows easy tradeoffs among space, time, and accuracy requirements. Moreover, SF-Tree can be *workload-adaptable*, and hence we can optimize our performance for answering frequently asked queries.

The rest of this paper is organized as follows. Section 2 describes the basic structure of an SF-Tree and analyzes the properties of SF-Trees. Various forms of SF-Trees are discussed in Section 3. The performance of SF-Trees is evaluated in Section 4. Finally, Section 5 concludes the paper.

2 SF-Tree

In this section we describe and analyze the structure of an SF-Tree. In an SF-Tree all SPEs in the documents are grouped into disjoint groups such that the SPEs in each group have the same selectivity[1]. Thus, we can find the selectivity of an SPE by finding to which group it belongs. Basically, SF-Tree is a tree structure that stores this grouping information efficiently and accurately. By efficiently we mean that SF-Trees take a small amount of space while supporting efficient retrieval. By accurately we mean that SF-Trees provide a *statistical accuracy guarantee* on the estimated selectivity.

[1] We can extend the idea of SF-Tree so that each group contains SPEs that have the same *quantized* selectivity. i.e., each group represents a *range* of selectivities.

2.1 Signature File

Before discussing the structure of an SF-Tree, let us first review a key technique in SF-Tree called signature file [4]. A signature file F is a bit vector that *summarizes* a set D of keywords (SPEs in our case) so that the existence of a keyword p_{query} in D can be checked efficiently. To create F, every keyword p in D is mapped by a hash function $h()$ into m bit positions and these bits are set to "1". To check the existence of a keyword p_{query} in D, we can check the m bit positions of $h(p_{query})$. If any of those bits is "0", p_{query} is *definitely* not in D. If all those bits are "1", p_{query} is *very likely* to be in D. However, there is a chance that those bits might be set by a combination of other keywords. In this case a *false drop* occurs and the probability for this to happen is called *false drop rate*. According to the analysis in [4], in order to minimize the false drop rate in a given space budget of $|F|$ bits, F, D, and m should be related by $|F| = m|D|/\ln 2$. With this condition, the false drop rate is $1/2^m$.

2.2 Structure of SF-Tree

An SF-Tree leaf node contains a *signature file* [4] that *summarizes* a group of SPEs with the same selectivity. The signature file in an internal node summarizes all SPEs in its descendant leaf nodes. The use of signature files increase the efficiency of storing the groups and determining whether an SPE is in a group. Figure 3 shows two example SF-Trees built from our invoice document. To retrieve the selectivity $\sigma(p)$ of an SPE p from an SF-Tree, we navigate from the root through all the internal nodes *containing* p to the leaf node that *contains* p. A node n *contains* an SPE p if p is in the group D of SPEs that are summarized by the node. Since every SPE has one and only one selectivity, p is contained in only one SF-Tree leaf node and all of its ancestors. However, since the SPEs in a group are summarized by a signature file, a node n may falsely report that it contains p. Therefore, this navigation may reach more than one leaf nodes though the probability is small. In this case a range of selectivities that can cover all the leaf nodes that contains p is reported because we cannot distinguish a false drop leaf node from a correct leaf node.

There are different types of SF-Trees because we impose no constraints on how the internal nodes are nested. Figure 3 shows two different SF-Trees. In the figure, g_k represents a group of SPEs that have a selectivity count of k. The selectivities of the SPEs are shown in Table 1.

2.3 Analysis on SF-Tree

We used *perfect* SF-Tree in our analysis but we can extend our analysis to other types of SF-Trees by similar arguments. A perfect SF-Tree has a form of a perfect binary tree and its leaves are ordered by their associated selectivities. We analyze the accuracy, selectivity retrieval time and storage requirements of SF-Tree in this section. You may refer to our technical report [6] for the proofs to the lemmas.

Accuracy An error on the retrieved selectivity $\sigma(p)$ occurs only when the retrieval algorithm reaches a leaf node n_f that falsely reports that it contains p. For this to happen, the ancestors of n_f must either have false drops or contain p. The following lemmas showed that either it is unlikely to occur or the error size is small.

Lemma 1. *Let p_- be an SPE query that does not appear in the document, d be the depth of a perfect SF-Tree T, and m be the number of bit positions in a signature file an SPE is mapped to. The probability ϵ_- for an error to occur in the estimated selectivity for a negative query p_- in a perfect SF-Tree is smaller than $1/2^{d(m-1)}$.*

Lemma 2. *Let p_+ be an SPE query that appears in the document, n_p be the leaf node that contains p_+, n_f be a false drop leaf node (i.e., a node that falsely reports that it contains p_+), and c be the difference in levels between n_f and the least common ancestor n_c of n_p and n_f. The probability ϵ_+ for a positive SPE p_+ to have error caused by n_f is smaller than $1/[2(2^{c(m-1)})]$.*

As shown in Lemma 1, the error rate decreases exponentially with respect to d and m. For $m = 8$, the error rate is less than $1/2^{16(8-1)} = 1/2^{112} \approx 0$ in a 16-level perfect SF-Tree for any p_-. Note that in Lemma 2, for a given c the number of leaf nodes in the subtree rooted at n_c is 2^c. The magnitude of error δ, which is the difference between the selectivity of n_f and n_p, is at most $2^c - 1$ because the selectivities of all leaf nodes are contiguous. Thus, by Lemma 2, errors with large magnitudes are unlikely to occur since the probability for an error to occur in a positive SPE p_+ is less than $1/2((\delta+1)^{(m-1)})$ which decreases exponentially with respect to the magnitude of an error δ.

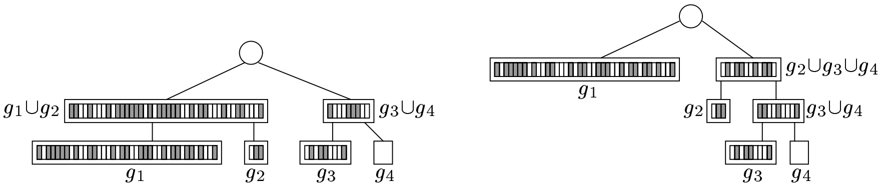
Selectivity Retrieval Time To retrieve the selectivity of a positive SPE p_+ from an SF-Tree T, we navigate from the root node of T to a leaf

	SPE (p)	$\sigma(p)$

g_1	`//@date, //buyer, //invoice, //products, //seller,`
	`//buyer/name, //seller/name, //invoice/@date,`
	`//invoice/buyer, //invoice/seller, //invoice/products,`
	`//invoice/buyer/name, //invoice/seller/name`
		1

g_2	`//name`	2
g_3	`//invoice/products/product, //products/product, //product`	3

Table 1. The selectivity of all the SPEs that appear in the invoice in Figure 1



(a) a perfect SF-Tree

(b) a Shannon-Fano SF-Tree

Fig. 3. Two SF-Trees for our invoice example.

node through the nodes that contains p_+. We need to check m bits in a signature file in order to test whether p is contained in a node n. Extra checkings may be required if false drop occurs since children of false drop nodes are also checked.

Lemma 3. *Let d be the depth of a perfect SF-Tree T. The expected number t of signature files to be checked to retrieve the selectivity of a positive SPE query p_+ from T is approximately* $2d + [(2d+2)(d+2)/2^m]$.

By Lemma 3, the expected number of bit positions to be checked depends only on d and m which are *independent* of the size of the document. It is time efficient to retrieve a selectivity from an SF-Tree especially when the document is huge. For a negative SPE query p_- that is not contained in the document, we usually only need to check the children of the root node before knowing that p_- is not contained in any nodes in the SF-Tree. Thus SF-Tree is even more efficient for retrieving the selectivity of negative SPE queries.

Storage Requirement In each node of an SF-Tree, two types of data are stored: tree pointers and a signature file. Since the space required by tree pointers in a tree is well understood, it is more interesting to analyze the space required to store all the signature files.

Lemma 4. *Let P be the set of all SPEs in the XML document, N be the set of all leaf nodes of an SF-Tree T, $n_p \in N$ be the leaf node that contains an SPE $p \in P$, $|n_p|$ be the number of SPEs contained in n_p, d_{n_p} be the depth of n_p in T, and m be the number of bit positions that an SPE is mapped to. The total space S required by all the signature files in an SF-Tree T is $(m/\ln 2)\sum_{n_p \in N} |n_p| d_{n_p}$.*

By lemma 4, $S = (m/\ln 2)\sum_{n_p \in N} |n_p| d_{n_p}$. The space requirement of an SF-Tree is linearly proportional to m, the number of bit positions that an SPE is mapped to, and the average weighted depth of the SF-Tree. Thus if we want to improve the space efficiency of an SF-Tree, we can adjust the value of m or the averaged weighted depth of the SF-Tree.

3 Variations of SF-Tree

We focused on perfect SF-Trees in the previous section because their simple and regular structures is easy to be analyzed. However, the structure of a perfect SF-Tree may sometimes be too regular that it may not fit the requirements of some users. Thus we describe various forms of SF-Trees to provide flexibility to fitting different user requirements. A more detailed discussion about various SF-Trees can be found in [6].

3.1 Shannon-Fano SF-Tree

As shown in Lemma 4, we can reduce the space requirement and improve the retrieval efficiency of an SF-Tree by reducing its average weighted depth. Huffman tree [7] is an obvious choice to minimize the average weighted depth of SF-Tree. However, since in a Huffman tree, groups with various selectivities may be combined, there is no relationship between c and the error size. Thus Lemma 2 can only give us a much looser accuracy guarantee for Huffman SF-Tree.

To preserve the tight accuracy guarantee, we used a *Shannon-Fano* [7] like method to construct our SF-Tree. We first sort all the leaf nodes according to their selectivities, and recursively divide the nodes into two sets of similar *weights* and contiguous selectivities. Figure 3 shows a Shannon-Fano SF-Tree. Since Shannon-Fano trees have comparable average weighted depth with Huffman trees and Shannon-Fano SF-Trees have a better accuracy guarantee, our experiments are based on Shannon-Fano SF-Trees instead of Huffman SF-Trees.

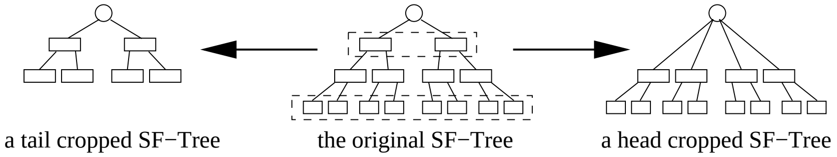


Fig. 4. Cropped SF-Trees

3.2 Cropped SF-Tree

Cropping an SF-Tree allows users to sacrifice accuracy and speed for space. By removing some layers (usually top or bottom) of an SF-Tree, the space required by an SF-Tree is reduced but the estimation accuracy is also reduced. We propose two types of cropped SF-Trees, *head-cropped* and *tail-cropped* SF-Trees. In a head-cropped SF-Tree, the layers of nodes around the root are removed. Children of the cropped nodes become children of the root. In a tail-cropped SF-Tree, the layers of nodes around the leaves are removed. The new leaves represent a range of selectivities of their deleted descendant leaves. The accuracy of a tail-cropped SF-Tree is lowered because we can only find a *range* of selectivities to which an SPE belongs. Figure 4 shows a head-cropped SF-Tree and a tail-cropped SF-Tree. The number of head and/or tail levels to be removed depends on how much selectivity retrieval speed and accuracy we could sacrifice. This facilitates flexible tradeoffs among space, time and accuracy.

4 Experiments

In this section we present the experimental evaluation on the performance of SF-Tree by using *Shannon-Fano* SF-Tree as an example. We compared SF-Tree with some previous approaches including path tree, global-* tree [1], and suffix trie [14]. We also evaluated the accuracy and the selectivity retrieval time of SF-Tree under different parameters that affect the space requirement. The parameters include m (the number of bit positions that an SPE maps to) and h (the number of top levels cropped). The experiments showed that SF-Tree is accurate and efficient over various ranges of parameters so that it is flexible. Due to space limitation, you may refer to [6] for the details of this experiment.

4.1 Experiment Setup

We used two different datasets, *XMark* and *MathML*, in our experiments. The XMark dataset, which models an auction web site, is generated by

	XMark	MathML
Number of Elements	206131	42688
Number of Diff. Tags	83	197
Size of Document	11.7MB	1.07MB
Number of Absolute Paths	537	39264
Number of Simple Paths	1912	228874

		XMark	MathML
Positive Queries	Number of Queries	2000	2000
	Average Query Length	5.94	6.793
	Average Selectivity	351	1.04
Negative Queries	Number of Queries	2000	2000
	Average Query Length	3.93	4.046
	Average Selectivity	0	0

Table 2. A summary of the properties of the datasets and the query workloads

the XML generator in the XML Benchmark project [13]. The MathML dataset is generated by the IBM XML Generator [3] using the DTD of MathML. Table 2 shows a summary of the properties of the two datasets. We used two query sets in our experiments. The *positive* query set contains 2000 randomly picked SPEs of the document and the *negative* query set contains 2000 random combinations of possible tags. Their properties are as shown in Table 2. We evaluated the accuracy (absolute and relative errors), selectivity retrieval time and space requirement of different Shannon-Fano SF-Trees.

4.2 Comparison with Previous Approaches

We compared the performance of SF-Trees with that of some previous approaches including *path trees*, *global-* trees* and suffix tries in our experiments. We compared them in terms of accuracy, selectivity retrieval time and storage requirement. Tables 3 and 4 show the result of this experiment.

Although we can only retrieve an estimated selectivity from an SF-Tree, by appropriately selecting m and h the estimates are very close to the exact values. As shown in Tables 3 and 4, we have *no* error on the estimated selectivity if $m = 8$ and $h = 3$. Even if we set $m = 6$ to save some space, the errors in the estimated selectivity is still very low. Thus the selectivity retrieved from an SF-Tree is very accurate in many parameter settings.

Global-* tree occupies the least amount of space compared with all the methods we tested. In our experiments, we asked the global-* tree to remove around 40% of the nodes in the path tree. We found that the selectivity estimated by a global-* tree has a relative error of around 200% on both datasets. It is because a global-* tree do not control the error induced into the estimated selectivity during its summarization process. Note that if the XML document is very complex and the selectivity of the path expressions does not vary a lot (as in the MathML dataset), SF-Tree can be smaller than a global-* tree.

	Path Tree	Global-* Tree	SF-Tree	SF-Tree	Suffix Trie
	(537 nodes)	(299 nodes)	($m = 8$, $h = 3$)	($m = 6$, $h = 3$)	
Averaged Abs. Error	0	146.2335	0	0.013	0
Averaged Rel. Errors	0	198%	0	0.025%	0
Storage Requirement	7.5KB	4.2KB	24KB	20.7KB	183KB
Speed up over Path Tree	1	3.98	3.27	3.68	32.28

Table 3. A Comparison of performance for the XMark dataset.

	Path Tree	Global-* Tree	SF-Tree	SF-Tree	Suffix Trie
	(61545 nodes)	(39590 nodes)	($m = 8$, $h = 3$)	($m = 6$, $h = 3$)	
Averaged Abs. Error	0	2.387	0	0.043	0
Averaged Rel. Errors	0	234%	0	4.3%	0
Storage Requirement	862KB	555KB	484KB	365KB	17.2MB
Speed up over Path Tree	1	4.22	2546	2704	5064

Table 4. A Comparison of performance for the MathML dataset.

Our experiments showed that SF-Tree is generally much more efficient than a path tree. Retrieving a selectivity of an SPE from an SF-Tree can be up to 2,704 times faster than retrieving the required selectivity from a path tree. While having such a large speed up, SF-Trees do not require much more space than path trees. Sometimes, an SF-Tree may require less space than a path tree.

In terms of selectivity retrieval time, suffix trie gives the best performance but it requires a large amount of space. Although SPE selectivity retrieval is less efficient in an SF-Tree than a suffix trie, an SF-Tree requires 10 to 50 times less space than a suffix trie. SF-Tree is thus a more space efficient method than suffix trie to achieve speed up over path trees.

5 Conclusions and Discussions

In this paper we proposed a new data structure called SF-Tree. An SF-Tree stores the statistics of an XML document using *signature files* and the files are organized in a *tree* form so that the selectivities of SPEs can be retrieved efficiently. Our analysis shows that an SF-Tree has a *statistical accuracy guarantee* on the selectivity retrieved from it and the error rate decreases exponentially with respect to the error size. Moreover, SF-Tree provides flexible tradeoffs between space, time and accuracy. Our experiments show that SF-Tree is much more efficient than path trees and much more space efficient than suffix tries.

We are investigating techniques for updating an SF-Tree. The selectivities of SPEs changes when the XML document is updated. Thus the updated SPE should change from one selectivity group to another. However, it is non-trivial since we only stored the summarized group information in a signature file.

We are also extending SF-Tree to other applications. Basically, SF-Tree is a new approach for storing a set of "key-to-value" pairs efficiently. In the selectivity estimation problem, we store the mapping between an SPE and its selectivity in an SF-Tree. Our analysis shows that SF-Tree is especially efficient when the domain of *key* is large but the domain of *values* is relatively small. Hence we can apply SF-Trees to other applications such as storing multi-dimensional histograms.

References

1. A. Aboulnaga, A. Alameldeen, and J. Naughton. Estimating the selectivity of XML path expressions for internet scale applications. In *VLDB*, pp. 591–600, 2001.
2. Z. Chen, H. V. Jagadish, F. Korn, N. Koudas, S. Muthukrishnan, R. Ng, and D. Srivastava. Counting twig matches in a tree. In *ICDE*, pp. 595–604, 2001.
3. A. L. Diaz and D. Lovell. XML data generator, Sept. 1999. `http://www.alphaworks.ibm.com/tech/xmlgenerator`.
4. C. Faloutsos and S. Christodoulakis. Signature files: An access method for documents and its analytical performance evaluation. *ACM TOIS*, 2(4):267–288, 1984.
5. R. Goldman and J. Widom. DataGuides: Enabling query formulation and optimization in semistructured databases. In *VLDB*, pp. 436–445, 1997.
6. W.-S. Ho, B. Kao, D. W. Cheung, YIP Chi Lap [Beta], and E. Lo. SF-Tree: An efficient and flexible structure for selectivity estimation. Technical Report TR-2003-08, The University of Hong Kong, Dec. 2003.
7. D. E. Knuth. *The Art of Computer Programming*, volume 3. Addison-Wesley, 1973.
8. Q. Li and B. Moon. Indexing and querying XML data for regular path expressions. In *VLDB*. pp. 361–370, 2001.
9. L. Lim, M. Wang, S. Padmanabhan, J. S. Vitter, and R. Parr. XPathLearner: an on-line self-tuning markov histogram for XML path selectivity estimation. In *VLDB*. pp. 442–453, 2002.
10. T. Milo and D. Suciu. Index structures for path expressions. In *ICDT 1999*, pp. 277–295, 1999.
11. N. Polyzotis and M. Garofalakis. Statistical synopses for graph-structured XML databases. In *SIGMOD*. pp. 358–369, 2002.
12. N. Polyzotis and M. Garofalakis. Structure and value synopses for XML data graphs. In *VLDB*. pp. 466–477, 2002.
13. A. R. Schmidt, F. Waas, M. L. Kersten, D. Florescu, I. Manolescu, M. J. Carey, and R. Busse. The XML Benchmark Project. Technical Report INS-R0103, CWI, Amsterdam, The Netherlands, Apr. 2001.
14. G. A. Stephen. *String Searching Algorithms*, volume 3 of *Lecture Notes Series on Computing*, chapter Suffix Trees, pp. 87–110. World Scientific, 1994.
15. W3C. Extensible markup language (XML) 1.0, Feb. 1998. `http://www.w3.org/TR/1998/REC-xml-19980210`.
16. W3C. XML path language (XPath) version 1.0, Nov. 1999.
17. W3C. XQuery 1.0: An XML query language, June 2001. `http://www.w3.org/TR/xquery`.