# Structural Selectivity Estimation for XML Documents

National ICT Australia Ltd., Technical Report PA006152, 2006.

D. Fisher
National ICT Australia Ltd. and
University of New South Wales
Sydney, Australia
damien.fisher@nicta.com.au

S. Maneth
National ICT Australia Ltd. and
University of New South Wales
Sydney, Australia
sebastian.maneth@nicta.com.au

## ABSTRACT

Estimating the selectivity of queries is a crucial problem in database systems. Virtually all database systems rely on the use of selectivity estimates to choose amongst the many possible execution plans for a particular query. In terms of XML databases, the problem of selectivity estimation of queries presents new challenges: many evaluation operators are possible, such as simple navigation, structural joins, or twig joins, and many different indexes are possible ranging from traditional B-trees to complicated XML-specific graph indexes. A new synopsis for XML documents is introduced which can be effectively used to estimate the selectivity of complex path queries. The synopsis is based on a lossy compression of the document tree that underlies the XML document, and can be computed in one pass from the document. It has several advantages over existing approaches: (1) it allows one to estimate the selectivity of queries containing all XPath axes, including the order-sensitive ones, (2) the estimator returns a range within which the actual selectivity is guaranteed to lie, with the size of this range implicitly providing a confidence measure of the estimate, and (3) the synopsis can be incrementally updated to reflect changes in the XML database.

## 1. INTRODUCTION

The Extensible Markup Language (XML) has found practical application in numerous domains, including data interchange, streaming data, and data storage. The semi-structured nature of XML allows data to be represented in a considerably more flexible nature than in the traditional relational paradigm. The tree-based data model underlying XML poses many challenges to efficient query evaluation.

An important component of any XML database system is effective *selectivity estimation*: given a query $Q$ over a database $\mathcal{D}$, what is the approximate result size of $Q$ over $\mathcal{D}$? This problem arises in several domains. Firstly, a rough estimate of the result size of a query can indicate to the user whether or not a query is appropriately framed before running a potentially expensive query. Selectivity estimation also has natural applications to approximate query answering. However, the most significant application of selectivity estimation is in query plan selection.

For example, suppose we have the sets $A$, $B$, and $C$ of all `a`, `b`, and `c` elements in a document, and we wish to evaluate the query //`a`[.//`b`]//`c`. We could do this by performing a structural join on $A$ and $B$, and joining this result with $C$. Alternatively, we could first join $A$ and $C$, and then join the intermediate result with $B$. The relative speed of these two queries is highly dependent on the selectivity of the initial structural joins. While for these kinds of queries a twig join is more appropriate, similar issues arise involving the relative result sizes for two or more twig queries, particularly in more sophisticated query languages such as XQuery.

Thus, in any database system, being able to accurately estimate the result size of the sub-expressions in a query is of great practical importance. There has been a lot of work on this problem in the context of XML databases [1, 6, 8, 12, 15–18, 24, 25, 27]. All previous work suffers from some combination of the following problems:

- *Expensive construction*: A problem with many techniques is that synopsis construction is extremely expensive. Any algorithm which requires more than one pass of the database is likely to be too expensive to run on very large databases.

- *Non-updateability*: Almost every selectivity estimation technique to date fails to handle updates to the underlying database. As they are static, their accuracy deteriorates as the database changes. The only realistic solution is to periodically rebuild them from scratch, which is obviously expensive.

- *Limited utility*: Selectivity estimation techniques generally consider only a limited subset of a query language. Most previous XML techniques consider *extremely* limited languages, such as simple path expressions. For example, no previous XML selectivity estimation technique can handle the order-sensitive axes of XPath, such as `following`.

- *No guarantee on accuracy*: All existing techniques use heuristics to generate their selectivity estimates. These heuristics, while based on well-justified assumptions in many cases, do not provide any guarantee of accuracy, and hence the computed estimate can be wildly inaccurate. With the exception of [25], no previous technique

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$5.00.

gives the user any sort of confidence measure on the result.

In this work, we extend recent work on the lossless compression of XML [5] to the problem of selectivity estimation. Our work has the following advantages over previous work:

- Our synopsis can be constructed in a single pass of the underlying document. As we shall see in our experiments, our construction cost is between 50 and 100 times faster than for other synopses.
- Our synopsis can give selectivity estimates for any Core XPath [9] query, including those which make use of order-sensitive axes.
- Unlike other selectivity estimation strategies, our approach returns a range within which the exact selectivity is *guaranteed* to lie. The confidence of the estimate is reflected in the size of the range: a smaller range naturally implies a greater degree of confidence in the answer. This is especially useful for query plan selection, as the query engine can take into account the confidence of the estimate when selecting plans.
- Our structure is efficiently updateable. While other structures require scans of the database to handle updates, we can handle updates in time linear in the size of the synopsis.

We demonstrate in our experimental section that even though our structure provides all these additional features, it returns answers that are competitive with the best existing techniques, while using an extremely small amount of space. Thus, our synopsis provides the first complete solution to the problem of selectivity estimation for structural queries.

The rest of the paper is organized as follows. In Section 4 we discuss our synopsis structure: *straight-line tree grammars*. In Section 5, we introduce the use of tree automata over these grammars as an effective means of computing selectivity. We then discuss how to efficiently update our synopsis in Section 6, as well as how to efficiently store it in Section 7. Section 8 presents an experimental analysis of our techniques, and Section 9 concludes the paper.

## 2. RELATED WORK

Aboulnaga et al [1] were the first to consider the selectivity estimation problem for simple path queries. They propose two different synopsis structures: pruned path trees and "Markov tables", which take advantage of the apparently Markovian nature of path selectivity in real-world XML data. The disadvantage of both approaches is that large structures must be constructed before they are pruned, which can be very space intensive; their experiments also demonstrate that their schemes have inconsistent performance. The idea of using a Markov table is extended to adaptive selectivity estimation by the XPathLearner system [12], which uses feedback from the query processor.

The first paper to study the problem of selectivity estimation for more complicated queries is that of Chen et al [6]; they use pruned suffix trees to estimate the selectivity of twig queries. The disadvantages of their approach are that, again, the whole suffix tree must be constructed before it is pruned, and also that their method does not generalize to handle the descendant operator of XPath. Freire et al [8] present a system, StatiX, which handles selectivity estimation in the presence of an XML Schema. Their work builds a histogram of data values for each element type in the schema; however, these histograms are built over the object identifiers of the nodes, which means that the quality of their estimation is highly dependent on the distribution of these identifiers. Ramanath et al [19] extended StatiX with updates, but their work still suffers from the same limitations.

Research upon the estimation of result sizes for the structural join operator, such as [24, 27], is also relevant to selectivity estimation for path expressions of the form $//p_1//p_2$. Unfortunately, these results cannot be easily generalized to other path expressions.

Polyzotis and Garofalakis [15–18] develop a general framework, XSketch, which provides good estimates for twig queries using graph synopses. The primary shortcoming of this method is that the construction process for the synopsis is expensive, and also relies on generating a set of test queries upon which the resultant synopsis is dependent.

All methods described above fail to handle updates on the underlying database. The only paper to consider selectivity estimation in a dynamic context is that of Wang et al [25], which makes use of Bloom filters to give provable guarantees on the quality of the results. However, these results only hold for simple path expressions of the form $/a/b$ or $//a/b$, and hence are of limited use in practice. Also, while the authors demonstrate the effectiveness of their approach, their technique requires the combination of two separate sketch structures, and the effect of the interaction of these structures on the estimation error is unclear.

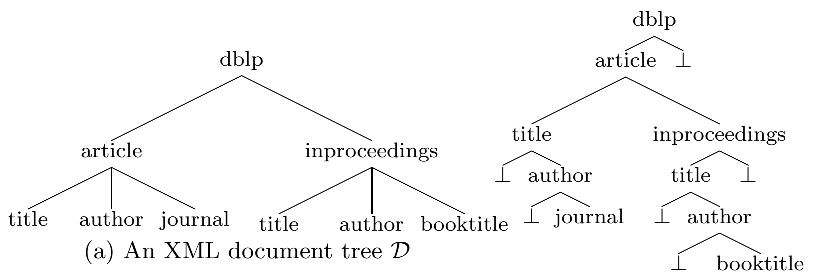
Sartiani [20, 21] has developed a general framework which can extend existing work on selectivity estimation to the problem of estimation for XQuery. As his technique is quite general, it can be applied to the work of this paper as well as existing work.

## 3. BASIC DEFINITIONS

**Documents** Let $\mathcal{D}$ be the ordered, rooted, labeled, unranked tree corresponding to an XML document; for our purposes we can safely ignore attributes, node values, namespaces, processing instructions, and other features of XML (many of these can be handled by our results in a straightforward fashion). By $\Sigma$ we denote the alphabet of elements present in $\mathcal{D}$; while in its full generality XML allows $\Sigma$ to be countably infinite in size, we restrict it for convenience so that it is finite and $|\Sigma| = O(1)$ (with respect to $|\mathcal{D}|$). Figure 1 gives an example of the structure of an XML document.

Throughout this paper we shall represent XML documents using a binary, ranked representation $bin(\mathcal{D})$ of $\mathcal{D}$. The transformation into this representation is simple: the left edge of the binary tree represents the "first child" relationship, while the right edge represents the "next sibling" relationship. We use $\perp$ to denote the empty tree, and write $V_{\mathcal{D}}$ for the vertices of the document (in the ranked representation), and $\lambda : V_{\mathcal{D}} \to \Sigma$ for the mapping from vertices of the document to their labels. Figure 2 gives an example of the transformation of an XML document from the unranked to ranked representation.

**Queries** Core XPath [9] is a powerful fragment of XPath that can be seen as the structural portion of XPath. It consists of queries satisfying the following grammar:



(a) An XML document tree $\mathcal{D}$

(b) The binary representation $bin(\mathcal{D})$

$$\begin{array}{rcl} A_0 & \rightarrow & \perp \\ A_1 & \rightarrow & title(\perp, author(\perp, y_1)) \\ Start = A_2 & \rightarrow & dblp(article(A_1(journal), \\ & & inproceedings(A_1(booktitle), \\ & & \perp)), \perp) \end{array}$$

(c) SLT grammar for $bin(\mathcal{D})$

**Figure 1: A sample XML document.**

$$\begin{array}{rcl} path & ::= & location\_path \mid / \; location\_path \\ location\_path & ::= & location\_step \; ( \; / \; location\_step \; )^* \\ location\_step & ::= & \chi :: t \mid \chi :: t \; [pred] \\ pred & ::= & (pred \vee pred) \mid (pred \wedge pred) \\ & & (\neg pred) \mid location\_path \end{array}$$

In this grammar, $\chi$ is an XPath axis (e.g., `descendant`, `descendant-or-self`, or `child`), and $t$ is a node test (i.e., either $t \in \Sigma$ or $t = *$). Note that the above grammar allows arbitrarily Boolean combinations of location paths as predicates in the query; for ease of presentation we will only consider conjunction in this paper (our results are easily generalized to handle other Boolean functions).

We will represent a core XPath query $Q$ as a tree with root $r_Q$, vertices $V_Q$ and edges $E_Q$, along with label functions $\lambda_V : V_Q \rightarrow \Sigma \cup \{*\}$ and $\lambda_E : E_Q \rightarrow A$, where $A$ is the set of XPath axes. Since $Q$ is a tree, each node $q$ in $Q$ has at most one parent – therefore, for convenience we write $\lambda_E(q) = \lambda_E(\langle \text{PARENT}(q), q \rangle)$. One of the vertices of $Q$, $m_Q \in V_Q$, is the *match node*. The semantics of an XPath query are well-known [7], and so we only briefly summarize them here. An embedding of a query $Q$ in a document $\mathcal{D}$ is a tree homomorphism $h : V_Q \rightarrow V_{\mathcal{D}}$ satisfying:

- $(\forall v \in V_Q)\ \lambda_V(v) = *$ or $\lambda(h(v)) = \lambda_V(v)$.
- $(\forall \langle v_1, v_2 \rangle \in E_Q)\ \langle h(v_1), h(v_2) \rangle$ satisfies the constraint specified by $\lambda_E(\langle v_1, v_2 \rangle)$.

The constraints specified by $\lambda_E$ depend on the axis, but are straightforward. For instance, if $\lambda_E(\langle v_1, v_2 \rangle) =$ `child`, then we require $h(v_1)$ to be the parent of $h(v_2)$ in $\mathcal{D}$. The result of the query $Q$ over $\mathcal{D}$ is then:

$$Q(\mathcal{D}) = \{h(m_Q) \mid \exists \text{ an embedding } h \text{ of } Q \text{ in } \mathcal{D}\}$$

The problem of selectivity estimation is to estimate $|Q(\mathcal{D})|$ for arbitrary queries $Q$.

While there are thirteen axes in XPath, several of these (e.g., `namespace`) are uninteresting as they can be handled in an analogous fashion to the others. The remaining axes can be divided into *forward* and *reverse* axes: in this paper, we only need to consider only the forward axes, as Olteanu et al [14] have demonstrated that any query involving reverse axes can be rewritten into one using only forward axes. Additionally, it is trivial to rewrite the `descendant` axis in terms of the `descendant-or-self` and `child` axes. Hence, we consider the axes `child`, `following-sibling`, `following`, `self`, and `descendant-or-self`. Note that this is purely a matter of convenience, as it is possible to extend our techniques to handle reverse axes more directly.

## 4. THE SYNOPSIS

The idea of our synopsis is to use a tree compression algorithm to generate a small pointer-based representation of the (ranked) tree $bin(\mathcal{D})$, called an "SLT grammar" (straight-line tree grammar). For common XML documents the size of the obtained grammar, in terms of the number of edges, is approximately 5% of the size of $\mathcal{D}$. We then decrease the size of this grammar further, by removing and replacing certain parts of it, according to a statistical measure of multiplicity of tree patterns. This results in a new grammar which contains size and height information about the removed patterns (this information will later be used to estimate selectivity). The two big advantages of SLT grammars over other compressed structures are: (1) they can be represented in a highly succinct way (see Section 7), and (2) they can be queried in a direct and natural way without prior decompression [13]. In particular, it is shown in Section 5 how to translate XPath queries into certain tree automata which can be executed on SLT grammars.

### 4.1 Tree Compression using SLT Grammars

Most XML documents are highly repetitive: the same tags appear again and again, and larger pieces of tag markup reappear many times in a document. One well-known idea of removing repeated patterns in a tree is to remove multiple occurrences of equal subtrees and to replace them by pointers to a single occurrence of the subtree. In this way, the minimal unique DAG (directed acyclic graph) of a tree can be computed in linear time. In [4] this idea was applied to XML document trees, and it was shown that for most document trees, the size of the minimal DAG is approximately 10% of the size of the original tree (where size is measured as the number of edges).

The idea of sharing common subtrees can be extended to the sharing of connected subgraphs in a tree. For example, in the tree $c(d(e(u)), c(d(f), c(d(a), a)))$ only the subtree $a$ appears more than once; however, the tree pattern "$c(d($" appears three times in the tree. The idea of sharing tree patterns gave rise to the notion of sharing graphs, which were studied in the context of optimal reductions of lambda-calculus [10]. The problem of finding a smallest sharing graph for a given tree is NP-complete. The first approximation algorithm for finding a small sharing graph is the BPLEX algorithm of [5]. Instead of sharing graphs, it produces isomorphic structures called *Straight-Line context-free Tree grammars (SLT grammars).* In such a grammar, a pattern is represented by a tree with *formal parameters* $y_1$, $y_2, \ldots$. For instance, the pattern "$c(d($" above is represented by the tree $c(d(y_1), y_2)$. Each nonterminal $A$ of the grammar has a fixed number $r(A)$ of formal parameters, called its rank. We call a finite set $N$ together with a rank mapping $r$ a *ranked alphabet.* A rule is of the form $A(y_1, ..., y_k) \to t$ where $t$ is a tree in which the formal parameters may appear at leaf nodes. We will only deal with grammars where each parameter appears exactly once in $t$.

Definition 1 (SLT Grammar). *An SLT Grammar $G$ (over $\Sigma$) is a tuple $\langle N, \Sigma, R\rangle$, where $N = \{A_1, \ldots, A_n\}$ is a ranked alphabet of nonterminals and $R$ is a set of rules. For each $A_i \in N$ of rank $k$ the set $R$ has exactly one rule of the form $A_i(y_1, \ldots, y_k) \to t$, where $t$ is a ranked tree over $\Sigma$, $N$, and $y_1, \ldots, y_k$, which are parameters appearing at the leaves of $t$, each exactly once, and in order (following the pre-order of $t$). Moreover, for any $A_j \in N$, if $A_j$ occurs in $t$ then $j < i$.*

The rules of $G$ are used as term rewriting rules in the usual way (inducing a rewrite relation $\Rightarrow_G$). The nonterminal $A_n$ is the start nonterminal. An SLT grammar $G$ produces (at most) one tree, because the indices of non-terminals strictly decrease (and hence no recursion is possible). For instance, the SLT grammar with rules:

$$\begin{array}{lll} A_1(y_1, y_2) & \to & c(d(y_1), y_2) \\ A_2 & \to & A_1(e(u), A_1(f, A_1(a, a))) \end{array}$$

generates the aforementioned tree. This can be seen by beginning with the start non-terminal $A_2$, and applying rules until no non-terminals remain:

$$\begin{array}{lll} A_2 & \Rightarrow_G & A_1(e(u), A_1(f, A_1(a, a))) \\ & \Rightarrow_G & A_1(e(u), A_1(f, c(d(a), a))) \\ & \Rightarrow_G & A_1(e(u), c(d(f), c(d(a), a))) \\ & \Rightarrow_G & c(d(e(u)), c(d(f), c(d(a), a))) \end{array}$$

The BPLEX algorithm is described in detail in [5]; In order for BPLEX to run in **linear time**, it is controlled by three parameters: the maximal rank that it gives to nonterminals, the maximal size of a pattern (= right-hand side), and the window size (= the number of rules that it scans when looking for existing patterns).

Let us explain how BPLEX works on an example. Every grammar generated by BPLEX contains the special nonterminal $A_0$ which generates the empty tree $\perp$. Running BPLEX on the tree $c(d(e(u)), c(d(f), c(d(a), a)))$ produces this SLT grammar:

$$\begin{array}{lll} A_0 & \to & \perp \\ A_1 & \to & a \\ A_2(y_1, y_2) & \to & c(d(y_1, y_2), A_0) \\ A_3 & \to & A_2(e(u, A_0), A_2(f, A_2(A_1, A_1))) \end{array}$$

Taking the binary encoding into account, this is basically the grammar that was shown before. BPLEX first looks for repetitions of subtrees and shares them by introducing nonterminals; in our example it introduces $A_1$ and replaces the two occurrences of $a$ by $A_1$. Next, BPLEX traverses this "DAG grammar" bottom-up searching for the repetition of a tree pattern consisting of at least two nodes; if it finds one, it introduces a new nonterminal, adds a corresponding rule, and replaces the occurrences of the pattern by the new nonterminal. In our case, when the second $c$ node is visited, coming from below, $A_2$ is introduced, and we obtain $A_2(f, A_2(A_1, A_1))$. When BPLEX moves further, it first looks for repetitions of patterns that are already in the grammar, and then looks for new patterns. At the root, it finds another occurrence of the $c(d($ pattern and replaces that by $A_2$.

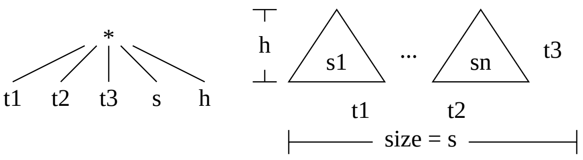
## 4.2 Lossy Compression

Consider an XML document tree $\mathcal{D}$ and an SLT grammar $G$ representing $bin(\mathcal{D})$. We want to reduce the size of $G$ in such a way that the result can be used for selectivity estimation. The idea is to keep the parts of the document tree that appear frequently (at many different positions in $bin(\mathcal{D})$), and to remove parts that appear infrequently. When we remove a part, we replace it by a special new symbol $*$, which additionally carries information about the height and size of the removed pattern. As parts we simply take the right-hand sides of the rules of $G$. Let $\kappa$ be a natural number, which we call the *threshold parameter.*

The threshold parameter determines how many productions will be removed (at most) from the grammar. The $A_0$-production is never deleted. First the productions with the lowest multiplicities are deleted, in the order $A_0, A_1, \ldots$ of the grammar; this process is repeated until $\kappa$ productions are deleted (or the grammar only contains the $A_0$-production). In this way we obtain a *($\kappa$-) lossy grammar (for $G$).* The *multiplicity* of $A_i$ is the number of times that $A_i$ is generated during the derivation of $bin(\mathcal{D})$ by $G$ (see below for how to compute this number). Deleting the nonterminal $A_i$ from $G$ means removing its rule $A_i(y_1, \ldots, y_k) \to t$, and (recursively) replacing in all other rules any subtree $A_i(t_1, \ldots, t_k)$ by the tree:

$$\begin{cases} *(t_1, \ldots, t_k, h, s) & \text{, if right-most leaf of ex}(t) \text{ is } y_k \\ *(t_1, \ldots, t_k, \perp, h, s) & \text{, otherwise} \end{cases}$$

where $h$ and $s$ are the height and size, respectively, of the unranked tree corresponding to the tree ex$(t)$ generated by $A_i$, i.e., the tree $u$ with $t \Rightarrow_G \cdots \Rightarrow_G u \not\Rightarrow_G$. The numbers $h$ and $s$ are stored to later give over-estimates of selectivity. Since we are working on binary trees, $u$ represents a sequence $\omega$ of trees in the unranked representation; if $y_k$ is the right-most leaf of ex$(t)$ then the last parameter tree $t_k$ of $A_i(t_1, \ldots, t_k)$ will be the last tree in $\omega$.



The (unranked) semantics of a tree $*(t_1, t_2, t_3, h, s)$ is depicted in the above figure: it represents any sequence $s_1, \ldots, s_{n+1}$ of trees such that: (1) the sequence $s_1, \ldots, s_n$ has subtrees $t_1$ and $t_2$, (2) the last tree $s_{n+1}$ equals $t_3$, (3) the height of the sequence is $h$, and (4) the size of the sequence is $s$. As an example, consider the grammar $G$ produced by BPLEX shown at the end of the previous subsection. Let us take $\kappa = 1$ and construct a $\kappa$-lossy grammar for $G$. The nonterminal $A_1$ will be deleted because it has the lowest multiplicity (= 2). Since $A_1$ generates $a$ which has size and height equal one, we replace occurrences of $A_1$ by the tree $*(1, 1)$. In this way we obtain the following grammar:

$$\begin{array}{lll} A_0 & \to & \perp \\ A_2(y_1, y_2) & \to & c(d(y_1, y_2), A_0) \\ A_3 & \to & A_2(e(u, A_0), A_2(f, A_2(*(1, 1), *(1, 1)))) \end{array}$$

As another example consider a nonterminal $A_5$, with rule $A_5(y_1, y_2) \to c(d(y_1, e), y_2)$ which is selected to be deleted.

If $A_7$ is *not* deleted and has the rule $A_7 \rightarrow A_3(A_5(A_2, A_1))$, then this rule will be changed in the lossy grammar into $A_7 \rightarrow A_3(*(A_2, A_1, 2, 3))$. However, if the $A_5$ rule was $A_5(y_1, y_2) \rightarrow c(d(y_1, y_2), e)$, then the $A_7$ rule would be changed into $A_7 \rightarrow A_3(*(A_2, A_1, \perp, 2, 3))$.

In order to replace nonterminals by stars, according to the value of $\kappa$, we must compute for each nonterminal its *multiplicity*, i.e., the number of times that it is generated during the derivation of the grammar. This can be done in one pass through the SLT grammar $G$ as follows: The nonterminal $A_n$ has multiplicity one. For each nonterminal that occurs $m \geq 1$ times in $t$, we set its multiplicity counter to $m$. We now move to the nonterminal $A_{n-1}$. If its multiplicity counter is $c \geq 1$ then for each nonterminal that appears $m' \geq 1$ times in $A_{n-1}$'s right-hand side we set the corresponding multiplicity counter to $c \cdot m'$. We proceed with $A_{n-2}, \ldots, A_1$ in the same way. Similarly, it is possible to compute the size of the tree that is generated by a given nonterminal. For computing the height of a right-hand side, for each occurrence of a nonterminal of rank $r \geq 1$ we must additionally take into account the lengths of the paths from the root of its right-hand side to the different parameter leaves.

## 5. SELECTIVITY ESTIMATION

In this section, we develop our selectivity estimation technique over SLT grammars. We first consider the conversion of an XPath query into an equivalent tree automaton, and describe how to evaluate this tree automaton over a document to test whether the query has at least one match in the document (i.e., whether the query accepts the document). We then extend the standard tree automaton to also return the size of the result of the query on a document. Then, the algorithm is generalized to work over SLT grammars. The final step is to handle lossy SLT grammars which contain $*$'s in rules.

Definition 2 (Tree Automaton). *A deterministic tree automaton over ranked tree encodings is a tuple $\langle P, \Sigma, \delta, F\rangle$, where $P$ is a finite set of states, $\Sigma$ is the alphabet, $\delta : P \times P \times \Sigma \rightarrow P$ is the transition function, and $F \subseteq P$ is the set of final states.*

A tree automaton is run on a tree in a bottom-up fashion as follows: the empty trees ($\perp$) which appear at the leaves of the tree are assigned the empty state, $\emptyset$. We then move upwards in the tree, so that a node with label $a$ whose children have been assigned states $p_1$ and $p_2$ is assigned the state $\delta(p_1, p_2, a)$. Once the automaton has reached the root node, and assigns it state $p_r$, we can determine whether the automaton accepts the document by testing whether $p_r \in F$.

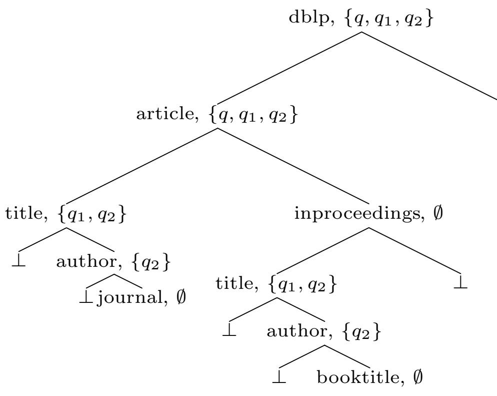
### 5.1 Converting Queries to Tree Automata

The translation of a core XPath query into a tree automaton is based upon the observation that core XPath queries can be evaluated in a bottom-up fashion on a document. For instance, consider the query $q$ = //`article`[.//`title`] [.//`author`]. This query can be decomposed into three subqueries: $q$ itself, $q_1$ = //`title`, and $q_2$ = //`author`. Working in a bottom-up fashion on the document tree in Figure 1, we can assign to each node in the database the subset of queries $\{q, q_1, q_2\}$ which match the subtree rooted at that node. This is easy to do, since, for instance, we know that

**Algorithm 1** Tree Automaton Transition Function

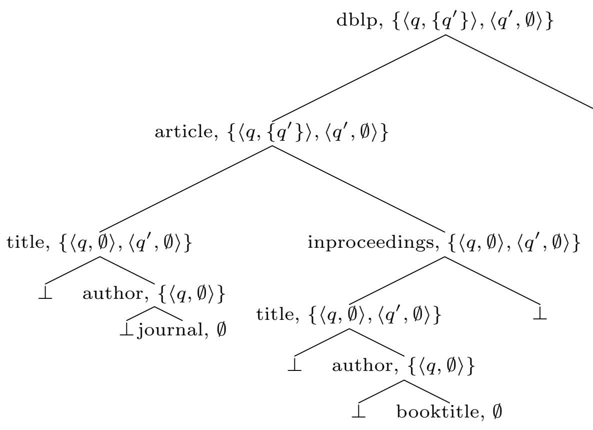
```
FOLLOWING ($n$)
  $R \leftarrow \emptyset$
  for each $c \in$ CHILDREN($n$) do
      if $\lambda_E(c)$ = following then
          $R \leftarrow R \cup \{c\}$
      else
          $R \leftarrow R \cup$ FOLLOWING($c$)
  return $R$

SATISFIED ($p_1, p_2, p, q, F$)
  for each $c \in$ CHILDREN($q$) do
      if $\lambda_E(c)$ = child $\wedge$ $\langle c, F \cap$ FOLLOWING($c$)$\rangle \notin p_1$ then
          return $false$
      else if $\lambda_E(c)$ = descendant-or-self $\wedge$
              $\langle c, F \cap$ FOLLOWING($c$)$\rangle \notin p \cup p_1$
          return $false$
      else if $\lambda_E(c) \in \{$following-sibling, following$\}$ $\wedge$
              $\langle c, F \cap$ FOLLOWING($c$)$\rangle \notin p_2$
          return $false$
      else if $\lambda_E(c)$ = self $\wedge$ $\langle c, F \cap$ FOLLOWING($c$)$\rangle \notin p$ then
          return $false$
  return $true$

$\delta$ ($p_1, p_2, a$)
  $F \leftarrow \{q \mid \langle q,$ FOLLOWING($q$)$\rangle \in p_2, \lambda_E(q)$ = following$\}$
  $p'_1 \leftarrow \{\langle q, (S \cup F) \cap$ FOLLOWING($q$)$\rangle$ |
          $\langle q, S\rangle \in p_1, \lambda_E(q) \in \{$descendant-or-self, following$\}\}$
  $p'_2 \leftarrow \{\langle q, (S \cup F) \cap$ FOLLOWING($q$)$\rangle$ |
          $\langle q, S\rangle \in p_2, \lambda_E(q) \in \{$following-sibling, following$\}\}$
  $p \leftarrow p'_1 \cup p'_2$
  for each $q \in \{q \mid q \in V_q, \lambda_V(q) \in \{a, *\}\}$ in post-order do
      if SATISFIED($p_1, p_2, p, q, F$) then
          $p \leftarrow p \cup \{\langle q, F \cap$ FOLLOWING($q$)$\rangle\}$
  return $p \cup p_2$
```

$q$ matches the document if both $q_1$ and $q_2$ match the left child, and if the label of the node is `article`. Recalling that we are working on the ranked, not unranked representation, the full calculation is:



The only axis which presents any significant difficulties is the `following` axis, as this introduces dependencies on nodes outside the subtree under consideration. For instance, consider the (contrived) query $q$ = //`author`/`following` :: `title`, which has the subquery $q'$ = //`following` :: `title`. Clearly, in Figure 1 the `author` node of the article lies in the result set of this query. In a bottom-up traversal of the query, however, this can only be determined once we reach the least common ancestor of this node and the `title` node of the `inproceedings` element (i.e., the root node of the document).

This problem can be addressed by keeping track not only of the matching sub-queries at each node, but also whether or not we have matched, for each of those nodes, any subquery which makes use of the `following` axis. Thus, instead of keeping track of subsets of $Q = \{q, q'\}$, we keep track of sets of items from $Q \times 2^Q$; the query accepts the document if $\langle q, \{q'\}\rangle$ lies in the set at the root. A run on the ranked representation of Figure 1 results in:



dblp, $\{\langle q, \{q'\}\rangle, \langle q', \emptyset\rangle\}$
article, $\{\langle q, \{q'\}\rangle, \langle q', \emptyset\rangle\}$
$\perp$
title, $\{\langle q, \emptyset\rangle, \langle q', \emptyset\rangle\}$
inproceedings, $\{\langle q, \emptyset\rangle, \langle q', \emptyset\rangle\}$
$\perp$
author, $\{\langle q, \emptyset\rangle\}$
$\perp$ journal, $\emptyset$
title, $\{\langle q, \emptyset\rangle, \langle q', \emptyset\rangle\}$
$\perp$
$\perp$
author, $\{\langle q, \emptyset\rangle\}$
$\perp$
booktitle, $\emptyset$

We now formalize this intuitive description. Given a query $Q$, our tree automaton has state set $P = 2^{Q \times 2^Q}$, final states $F = \{p \mid p \in P, \langle r_Q, \text{Following}(r_Q)\rangle \in p\}$ (see Algorithm 1 for Following), and transition function as given in Algorithm 1. Much of the complexity in Algorithm 1 is simply due to the differences in handling the semantics of the different axes, especially `following`.

## 5.2 Counting with Tree Automata

Once we have a tree automaton for a query $Q$, testing whether there is a match for $Q$ in a given document is straightforward, as we have seen above. However, in the context of selectivity estimation, we do not want to test acceptance, but instead want to return the size of result of the query. Seidl [23] developed a framework for finite tree automata with cost functions which addresses such problems: each transition in the automaton is assigned a cost, and the task is then to find the "cheapest" accepting path. We will use a similar technique here.

When running our automaton on a document, we must now keep more information in each state, in order to keep track of selectivity. To this end, we associate with each state $p$ in our automaton a set of counters. Our annotated state $\langle p, C\rangle$, consists of a normal state, $p \in 2^{Q \times 2^Q}$, and an array of counters, so that $C[\langle q, F\rangle]$ is the counter for each $\langle q, F\rangle \in p$. We will assume that $C[\langle q, F\rangle] = 0$ if $\langle q, F\rangle \notin p$. Each counter represents the number of nodes matching the corresponding subquery, that have not already been matched by that subquery's parent query. As we move up the query tree, we can use the counters for the sub-queries to compute the selectivity of each query node.

In order to maintain selectivity information, we extend the transition function to handle annotated states, as shown in Algorithm 2. The counting is relatively straightforward — we count the number of nodes matching the match node of the query, and propagate these counts up the query tree as required. Figure 2 gives an example run of our algorithm (since there are no `following` axes present in the query, we have used the simpler state type).

There are two issues that are worth mentioning. When we match a new query node, the match count for that query node is clearly the sum of the counts of its children. How-

**Algorithm 2** Count-Automaton Transition Function

```
RESTORE-COUNTS $(\langle p, C\rangle, \langle p_1, C_1\rangle, p'_1)$
1:  for each $\langle c, F\rangle \in p_1 \backslash p'_1$ do
2:      if $m_Q$ is a descendant of $c$ then
3:          Let $c \rightarrow q_1 \rightarrow \ldots \rightarrow q_n = m_Q$ be the path from
              $c$ to $m_Q$
4:          for $i$ from 1 to $n$ do
5:              if $\langle q_i, F \cap \text{Following}(q_i)\rangle \in p$ then
6:                  $C[\langle q_i, F \cap \text{Following}(q_i)] += C_1[\langle c, F\rangle]$
7:                  break

$\delta$ $(\langle p_1, C_1\rangle, \langle p_2, C_2\rangle, a)$
1:  $F \leftarrow \{q \mid \langle q, \text{Following}(q)\rangle \in p_2, \lambda_E(q) = \texttt{following}\}$
2:  $p'_1 \leftarrow \{\langle q, (S \cup F) \cap \text{Following}(q)\rangle \mid$
        $\langle q, S\rangle \in p_1, \lambda_E(q) \in \{\texttt{descendant-or-self}, \texttt{following}\}\}$
3:  $p'_2 \leftarrow \{\langle q, (S \cup F) \cap \text{Following}(q)\rangle \mid$
        $\langle q, S\rangle \in p_2, \lambda_E(q) \in \{\texttt{following-sibling}, \texttt{following}\}\}$
4:  $p \leftarrow p'_1 \cup p'_2$
5:  Initialize counter array $C$ for $p$ from $p'_1$ and $p'_2$
6:  for each $q \in \{q \mid q \in V_q, \lambda_V(q) \in \{a, *\}\}$ in post-order do
7:      if SATISFIED$(p_1, p_2, p, q, F)$ then
8:          $p \leftarrow p \cup \{\langle q, F \cap \text{Following}(q)\rangle\}$
9:          $C[\langle q, F \cap \text{Following}(q)\rangle] \leftarrow$
              $\sum_{c \in \text{Children}(q)} C[\langle c, F \cap \text{Following}(c)\rangle]$
10:         if $q = m_Q$ then
11:             $C[\langle q, F \cap \text{Following}(q)\rangle] += 1$
12:         $(\forall c \in \text{Children}(q))$ $C[\langle c, F \cap \text{Following}(c)\rangle] \leftarrow 0$
13: RESTORE-COUNTS$(\langle p, C\rangle, \langle p_1, C_1\rangle, p'_1)$
14: $p \leftarrow p \cup (p_2 \backslash p'_2)$
15: Update counter array $C$ for $p$ from $p_2 \backslash p'_2$
16: return $\langle p, C\rangle$
```

ever, once we have copied over the children counts, we must zero them out as well. This is to prevent double-counting, which occurs when multiple embeddings of a query in the document yield the same match node. For instance, in Figure 2(b), the node $c_1$ is matched by two embeddings of the query in Figure 2(a); at node $b_2$, however, we set $q_4 : 0$ in order to count only one embedding. The second (related) issue can be seen in the transition from the element $b_2$ to the element d in the document. Since the parent of $b_2$ does not have label a, $q_2$ is no longer a matching subquery — however, its child, the subquery $q_3$, *is* a matching subquery. Therefore, when removing $q_2$ from the set of matching subqueries, we must transfer its count of matching nodes back to $q_3$. This is the purpose of the function Restore-Counts in Algorithm 2.

## 5.3 Tree Automata over SLT Grammars

Up till this point we have considered tree automata running over a document. In this section, we demonstrate how to evaluate tree automata directly over SLT grammars, so that we can compute selectivity in time proportional to the size of the SLT grammar used to represent the document. Since this is much smaller than the document, it provides a feasible way of determining selectivity.

Again, we first consider the problem of acceptance, instead of selectivity computation. The main obstacle to running tree automata over SLT grammars is the handling of parameters in rules — since these can represent anything, we do not know what states they will take when evaluating the automaton on a rule.

The natural solution is to simply compute all possibilities. If we are considering a rule $A_i(y_1, \ldots y_k) \rightarrow t$, then we can define a function $\sigma_i(p_1, \ldots, p_k) \rightarrow P$ which gives the state for $t$, assuming the parameters map to the states $p_1, \ldots p_k$. In defining the function $\sigma_i$, we will need to make calls to the

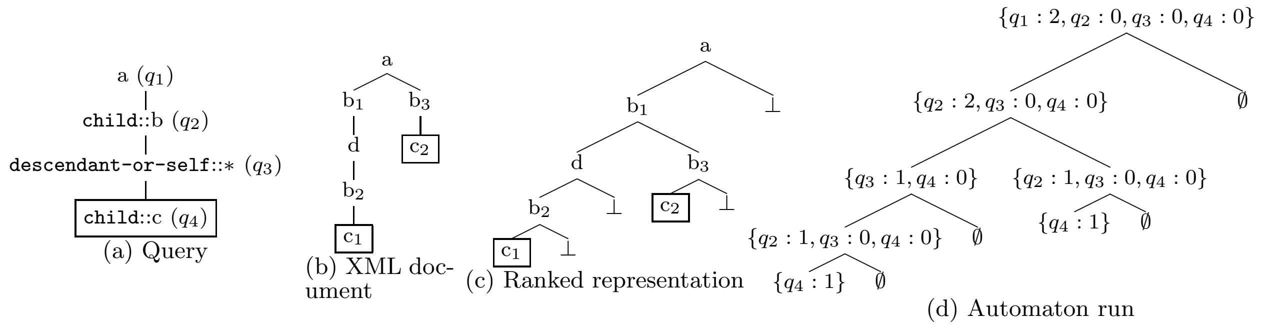


**Figure 2: Counting selectivity using tree automata.**

functions $\sigma_j$ for all rules that the rule for $A_i$ makes use of — however, at that point we know exactly what states to pass in as parameters to these functions.

Extending this to selectivity counting poses an additional problem: when computing the result of a query, one must incorporate the selectivity counts from the parameters. This can be done by manipulating the counters for parameter states symbolically. As can be seen in Algorithm 2, we only ever perform additions and the zeroing out of counters. Thus, if we treat the counters of the states corresponding to each parameter as unknown variables, then the selectivity count for the rule will be a linear function over these counters. This function, $f_i(p_1, \dots p_k) \to \mathbb{Z}$, can be determined by a natural extension to Algorithm 2, and hence it is easy to extend $\sigma_i$ to also compute $f_i$. When we come across a non-terminal $A_i(t_1, \dots, t_k)$ in the right hand side of a rule, we can compute its state by first recursively determining the states $p_1, \dots p_k$ corresponding to the input parameters, and using these and $\sigma_i$ to determine the corresponding state (and selectivity counts) for the non-terminal.

Figure 3 demonstrates the computation and use of the functions $\sigma_i$ when evaluating the query of Figure 2(a) over an SLT grammar for Figure 2(c). We run through the rules in a bottom-up fashion (from $A_0$ to $A_3$), using the functions $\sigma_j$ for $j < i$ to compute the state for rule $A_i$. For rules with parameters, we do not compute all possible values for the functions $\sigma_i$, but only those that are actually needed. This can be most easily seen by considering a *top-down* run through the grammar: we begin with rule $A_3$. To compute the state for the root node, we first need to compute the states for its two children, which are the two grammar fragments $A_1(d(A_2, A_0), A_2)$ (corresponding to the subtree at $\mathrm{b}_1$) and $A_0$ (corresponding to the empty tree). To compute the state for the first fragment, we must then compute the value of $\sigma_1(d(\sigma_2, \sigma_0), \sigma_2)$; this computation is deferred until we know the values of $\sigma_2$ and $\sigma_0$. We continue by recursing top-down, using dynamic programming to ensure that we do not evaluate the same value twice. As can be seen in Figure 3(b), the function $\sigma_1$ only needs to be computed for two different argument values.

Determining complexity is straightforward. If every rule has at most $k$ parameters, then we have:

THEOREM 3. *Selectivity counting over a straight-line grammar $G$ with $k$ parameters by a deterministic tree automaton with state set $P$ takes time $O(|P|^k|G|)$.*

It is worthwhile relating the size of the state set $P$ back to the size of a query. Clearly, $|P| = O(2^{2^{|Q|}})$, but in practice $|P|$ is *much* smaller. If we assume there are no `following` axes present in the query, then we can make this observation: if a node $q$ lies in a state $p$, then all of $q$'s descendants in the query also lie in $p$. This means that if we have a query which has at most $b$ branches, then there are only $(|Q|/b)^b$ different possible states. If we also have $m$ `following` axes in the query, then these increase the number of states by a factor of $2^m$. Therefore we have:

THEOREM 4. *Determining acceptance of a straight-line grammar $G$ with $k$ parameters by a query with branching factor $b$ and $m$ `following` axes takes time $O\left(\left(\frac{|Q|}{b}\right)^{bk} 2^{mk}|G|\right)$.*

In practice, BPLEX returns very small grammars even with a very low value for $k$ (such as $k \leq 2$), and so we can ignore this dependency. Also, the branching factor of queries is usually quite low, and we suspect that the occurrence of `following` axes in queries is infrequent. Finally, note that we do not need to explicitly compute all possible values for the functions $\sigma_i$, but instead can lazily compute only those values needed: we find that in practice only a small number of combinations of states are seen, and so this algorithm runs quickly. In Figure 3, e.g., only 2 out of 16 possible values for the function $\sigma_1$ are computed. Thus, the worst case bounds are generally not reached in practical situations.

## 5.4 Tree Automata over Lossy Grammars

Running tree automata over lossy SLT grammars is identical to running them over SLT grammars, except for the handling of $*$ nodes. In this case, we provide two alternative mechanisms for computing selectivity. These two methods lead to lower and upper bounds on the actual selectivity.

**Lower Bounds** The most straightforward approach to handling $*$ nodes is to simply ignore them — since this means we miss some nodes in the underlying database, computing selectivity in this fashion necessarily leads to a lower bound on the actual selectivity. In this case, the transition function can be easily given in terms of the function of Algorithm 2. For a $*$ subtree $*(t_1, t_2, \dots, t_k, h, s)$, it suffices to run the transition function already given on a tree of the form $*(*(\dots(*(*(t_1, t_2), t_3)\dots), t_{n-1}), t_n)$.

**Upper Bounds** Estimating upper bounds is straightforward conceptually, but the details are quite involved, and hence an exact technical description is omitted. The basic idea is that when the tree automaton reaches a $*$ node, it must consider all possible trees that the $*$ node could have replaced, subject to the height and size

$$
\begin{array}{rcl}
A_0 & \rightarrow & \perp \\
A_1(y_1, y_2) & \rightarrow & b(y_1, y_2) \\
A_2 & \rightarrow & A_1(c(A_0, A_0), A_0) \\
\mathit{Start} = A_3 & \rightarrow & a(A_1(d(A_2, A_0), A_2), A_0)
\end{array}
$$

(a) SLT grammar $H$ for Figure 2(c)

$$
\begin{array}{rcl}
\sigma_0 & = & \emptyset \\
\sigma_1(\{q_4 : c_1\}, \emptyset) & = & \{q_2 : c_1, q_3 : 0, q_4 : 0\} \\
\multicolumn{3}{l}{\sigma_1(\{q_3 : c_1, q_4 : c_2\}, \{q_2 : c_3, q_3 : c_4, q_4 : c_5\})} \\
& = & \{q_2 : c_1 + c_3, q_3 : c_4, q_4 : c_2 + c_5\} \\
\sigma_2 & = & \sigma_1(c(\sigma_0, \sigma_0), \sigma_0) \\
& = & \sigma_1(\{q_4 : 1\}, \emptyset) \\
& = & \{q_2 : 1, q_3 : 0, q_4 : 0\} \\
\sigma_3 & = & a(\sigma_1(d(\sigma_2, \sigma_0), \sigma_2), \sigma_0) \\
& = & a(\sigma_1(\{q_3 : 1, q_4 : 0\}, \{q_2 : 1, q_3 : 0, q_4 : 0\}), \emptyset) \\
& = & \{q_1 : 2, q_2 : 0, q_3 : 0, q_4 : 0\}
\end{array}
$$

(b) Computation of state functions on $H$

**Figure 3: Counting selectivity using tree automata over SLT grammars.**

constraints. It is possible to do this in time linear in the height of the replaced tree. Due to the flat nature of real world XML, the height of the replaced tree is very small, and this imposes significant constraints on the possibilities. Even in the event that there are many possible trees, the total contribution from a $*$ node to the selectivity estimate is bounded above by the number of nodes in the tree it replaced.

There is one optimization which we found boosted the accuracy of the upper bounds generated by our scheme considerably. For each element label $a \in \Sigma$, it is trivial to compute the set of element labels that occur as children of elements $a$ in the XML document. This information, which adds very little to the overall space cost of the synopsis, can be used to prune the number of possibilities in a $*$ node considerably. For instance, if we know that the set of possible children of an element $a$ are $\{b, c\}$, and if we are considering a $*$ node that is a child of an $a$ element, then the root node of the tree replaced by the $*$ node must have been labeled either $b$ or $c$. We can apply this procedure recursively up to the height bound $h$; when combined with the fact that the query often only involves a handful of unique element labels, this can have a dramatic effect on the quality of the upper bound estimates.

## 6. INCREMENTAL UPDATES

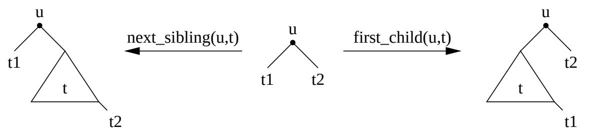
To date, the update problem has not yet been considered for SLT grammars. In this section, we present an effective update algorithm for lossless SLT grammars. We then extend our synopsis structure to a two-layer data structure:

- The lossy synopsis structure we have presented so far is stored, in a compact form in memory (as described in Section 7).
- An equivalent lossless SLT grammar is stored on disk.

When updates occur in the database, we update the grammar using the algorithms presented in this section. In order to minimize disk accesses, we can queue up updates to the structure, thus letting it get out of date for short periods of time. Once a sufficient number of updates to the grammar has occurred, we can recompute the in-memory synopsis in a single pass over the disk-based grammar. Since the disk-based grammar is still substantially smaller than the complete document, we can construct a new lossy synopsis quickly.

Clearly, we do not want to decompress the grammar into the document tree, do the update there, and then compress it back into a grammar. Instead, we would like to have an *incremental* way of doing updates directly on the grammar. As it turns out, incremental updates are surprisingly easy to support for SLT grammars: the idea is to rewrite the right-hand side $t$ of the start rule $A_n \rightarrow t$ of the grammar, until the node at which the update shall occur is "terminally" available; the latter means that the path from the root to this node does not contain any nonterminals. At this moment we know that the current node is not shared by other nonterminals and therefore the update can be carried out at the node.

The update operations we consider are: the insertion of a new tree as the first child of a node, as the next sibling of a node (which means as the right child in our ranked setting), and the deletion of a subtree (which means the deletion of a node and its left subtree in the ranked setting). The effect of an insertion as the first child and as the next sibling of a node $u$ in a ranked tree is shown in the following figure.



After an update has been realized on the (partially rewritten) right-hand side of the start rule, we run the BPLEX compression on this tree, replacing patterns that already appear as right-hand sides in the grammar by corresponding nonterminals, and possibly introducing new rules for newly found patterns that appear multiple times. As we will see in the experimental section, updates done in this way do not increase the size of the grammar significantly, and the increase in size stays constant even as the number of updates increases; hence, we never have to go back to the database and recompute a new grammar from scratch. Obviously, only linear time is needed for an update.

THEOREM 5. *The insertion of a tree $t$ into $\mathcal{D}$ can be realized on the SLT grammar $G$ (for $bin(\mathcal{D})$) in time $O(|G|+|t|)$, and the deletion of a subtree of $\mathcal{D}$ in time $O(|G|)$.*

More concretely, we use three different update operations.

Clearly they suffice to express any form of update to the document tree. The operations are:

- *first_child bindd_path tree*
- *next_sibling bindd_path tree*
- *delete bindd_path*

where *bindd_path* is a node in the ranked document tree in binary dotted decimal notation (Dewey notation), and *tree* must be a tree with right-most leaf $\perp$. Note that $bin(W)$ for any sequence $W$ of document trees is always of this form. The set of nodes $Dewey(t)$ in bindd notation of a binary tree $t = c(t_1, t_2)$ is $\{\varepsilon\} \cup \{i.d \mid d \in Dewey(t_i), i \in \{1,2\}\}$, and $\{\varepsilon\}$ if $t = d$ for some symbol $d$ of rank zero, where $\varepsilon$ denotes the empty sequence. We use binary Dewey notation since it can be easily derived from a normal Dewey encoding; however, it is important to note that this is only one possible means of linking between nodes in the database and nodes in the synopsis. An alternate strategy would be to label each node in the synopsis with a unique identifier and have nodes in the database point to the node in the synopsis within which they lie. The method of linking between the database and any index or synopsis structure is obviously highly implementation dependent, but such a mechanism is required for any updateable structure.

As an example, consider the grammar previously used:

$$\begin{array}{lll} A_0 & \to & \perp \\ A_1 & \to & a \\ A_2(y_1, y_2) & \to & c(d(y_1, y_2), A_0) \\ A_3 & \to & A_2(e(u, A_0), A_2(f, A_2(A_1, A_1))) \end{array}$$

and the update operation *delete* 1.2.1. We have to rewrite the right-hand side of the start rule until the node 1.2.1 has no nonterminals on its path to the root. We apply the rule for $A_2$ at the root node and rewrite:

$$A_2(e(u, A_0), A_2(f, A_2(A_1, A_1))) \Rightarrow_G \\ c(d(e(u, A_0), A_2(f, A_2(A_1, A_1))), A_0).$$

Since node 1.2 is still nonterminal, we rewrite it and obtain: $c(d(e(u, A_0), c(d(f, A_2(A_1, A_1)), A_0)), A_0)$. Now the node 1.2.1 is terminally available and can be deleted. We obtain the tree $c(d(e(u, A_0), c(A_2(A_1, A_1), A_0)), A_0)$. Finally, BPLEX is run on this right-hand side, i.e., it searches for existing and new patterns. When BPLEX reaches the root node of the tree, it detects the pattern $c(d(y_1, y_2), A_0)$ which exists as right-hand side of $A_2$. It replaces it and we obtain the tree $A_2(e(u, A_0), c(A_2(A_1, A_1), A_0))$. The final grammar after the update is:

$$\begin{array}{lll} A_0 & \to & \perp \\ A_1 & \to & a \\ A_2(y_1, y_2) & \to & c(d(y_1, y_2), A_0) \\ A_3 & \to & A_2(e(u, A_0), c(A_2(A_1, A_1), A_0)) \end{array}$$

Next, we consider an insertion (on the original grammar). We want to insert the tree $e(u)$ as first child of the second $d$ node, i.e., we want to execute *first_child* 1.2.1 $e(u)$. As for the delete, we first rewrite the start right-hand side until no nonterminals are on the path to the root node. As before, we obtain $c(d(e(u, A_0), c(d(f, A_2(A_1, A_1)), A_0)), A_0)$. Now we insert $e(u)$ as the new first child of the second $d$ node. We get $c(d(e(u, A_0), c(d(e(u, f), A_2(A_1, A_1)), A_0)), A_0)$. Finally, we run BPLEX on this tree. This time it discovers a new pattern $e(u, y_1)$ that appears twice; it therefore adds the new nonterminal $A_3$. The final grammar after update is:

$$\begin{array}{lll} A_0 & \to & \perp \\ A_1 & \to & a \\ A_2(y_1, y_2) & \to & c(d(y_1, y_2), A_0) \\ A_3(y_1) & \to & e(u, y_1) \\ A_4 & \to & A_2(A_3(A_0), A_2(A_3(f), A_2(A_1, A_1))) \end{array}$$

The *next_sibling* update works analogously to *first_child*, and hence an example is omitted.

## 7. SUCCINCT SYNOPSIS STORAGE

At this point, we have an SLT grammar $G$, which has already been made lossy, and hence has $*$ nodes. The natural in-memory representation of such a structure is to have a list of rules, with the right-hand side of each rule stored in a pointer-based tree data structure. However, this representation provides substantially more power than we really need: a pointer-based tree structure allows access to (child/sibling) nodes in constant time. Since a bottom-up tree automaton can be easily implemented by a depth-first, left-to-right tree traversal, we only need to have constant time access to the root node of the right-hand side of each rule. Thus, we can compress the synopsis considerably by using a more sophisticated representation. In this section, we will first consider the case of a static synopsis, and then extend this data structure to allow efficient updates.

**The Static Case** We recall the following properties of our synopsis and estimation algorithm:

- When evaluating a rule $R_i = A_i \to t$ of $G$, the estimation algorithm only needs to access rules $R_j$ where $j \leq i$.
- When evaluating a rule $R$, the estimation algorithm runs through the right-hand side in a single post-order traversal.
- For a rule $R$ with $k$ parameters, each parameter is used only once, and the parameters appear sequentially in a pre-order traversal of the right-hand side of $R$.

We take advantage of these properties to construct a packed representation for the synopsis. For each rule $R$, we construct a packed bit encoding $E(R)$, and then encode the entire synopsis as the concatenation $E(R_0) \cdot E(R_1) \cdot \ldots \cdot E(R_n)$. When running a tree automaton over the synopsis, we start by decoding the first rule, $R_0$. Once we have decoded rule $R_0$, we know where rule $R_1$ starts; more generally, once we have decoded rule $R_i$, we know where rule $R_{i+1}$ starts. Since the tree automaton runs in a bottom-up fashion, when it has reached rule $R_i$ it will have all the information necessary to process this rule, as long as it remembers the start locations of all the rules it has seen up to that point (needed for the "lazy computation" described at the end of Section 5.3).

In addition to the packed representation above, we maintain a lookup table to further reduce the size of the representation of $*$ subtrees. Recall that each $*$ node has associated with it two statistics, the height $h$ of the replaced tree, and the number $s$ of nodes replaced. We construct an array $S[i]$ consisting of all unique tuples $\langle h, s\rangle$ (since $*$ nodes often replace patterns that occur more than once, it is likely that a fixed $\langle h, s\rangle$ will occur more than once in the grammar). When we reach a $*$ node, we can use the appropriate offset into this array instead of explicitly listing $h$ and $s$.

For a rule $R_i$ with $k$ parameters, we construct $E(R_i)$ as follows: first, add $k$ ones followed by a zero bit to encode the parameter count. Following this we encode the right-hand side of the rule as a list of symbols, which give its pre-order traversal. There are four possibilities for the first symbol:

| $A_2(y_1, y_2) \rightarrow$ | $a($ | $A_1($ | $b($ | $y_1$ | $\_)),$ | $*($ | $y_2,$ | $c($ | $\_,$ | $\_)$ | , | $5, 10)$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 110 | 010 | 110 | 011 | 001 | 101 | 000 | 1001 | 1100 | 101 | 101 | 0 | 0 |

| **Symbol Table** | | | **$*$ Statistics** | |
|---|---|---|---|---|
| $* \rightarrow 0$ | $a \rightarrow 2$ | $A_0 \rightarrow 5$ | 0 | $h = 5, s = 10$ |
| $y_i \rightarrow 1$ | $b \rightarrow 3$ | $A_1 \rightarrow 6$ | | |
| | $c \rightarrow 4$ | | | |

**Figure 4: Packed encoding for a rule.**

- A call to a rule $R_j(t_1, t_2, \ldots, t_k)$, $j < i$: there are $i - 1$ possible rules that can be called from $R_i$.
- A terminal $a(t_1, t_2)$: there are $|\Sigma|$ possibilities.
- A star or a parameter: in each case, there is only one possibility.

Thus, we can encode all possibilities in $\log(|\Sigma| + i + 1)$ bits. The remaining encoding then depends on each of the possibilities:

- For a rule $R_j(t_1, t_2, \ldots, t_k)$ or terminal $a(t_1, t_2)$, we simply recurse the encoding algorithm on each subtree $t_1, t_2, \ldots t_k$, and store the concatenation of these encodings. In both cases we know exactly how many parameters there are, and hence do not need to encode this information.
- For a $*$ subtree $*(t_1, \ldots, t_k, h, s)$, the number of parameters is variable. Therefore, in addition to storing the appropriate offset into the lookup table $S$, we must store the encodings of the $k$ subtrees, as well as $k$ itself. One way of doing this is to prefix the encoding of each subtree with a single 1 bit, and terminate the list of parameters with a 0 bit.

Figure 4 gives the encoding for a sample rule. This simple scheme slashes the space requirements for a synopsis. A variable length encoding for symbols further improves space usage. Note that the ability to encode our structure in this way does not apply to other XML synopses, such as XSKETCH, because in those structures each node can be pointed to by any other node, and thus a pointer-based representation is necessary.

**The Dynamic Case** In the dynamic case, for small synopses it is easy to simply re-encode the entire synopsis from scratch. For larger synopses, we split the encoding into an array of blocks, leaving padding in each block. A standard ordered file maintenance algorithm, such as that of Bender et al [3] can then be used to speed up insertions and deletions (for an array of $n$ elements, we can insert and delete elements maintaining the order of the array in $O(\log^2 n)$ time).

## 8. EXPERIMENTS

In this section we give an empirical evaluation of our system. Our experiments were implemented in C and C++. For simplicity, we did not implement our packed representation, since it does not affect the quality of our results. The synopsis sizes we report in this section assume that the packed representation was used. The BPLEX algorithm was used with maximal rank 10, maximal size of a right-hand side 20, and window size 40000 (1000 in the case of updates), cf. end of Section 4.1 for an explanation of these parameters.

| Data Set | Size (MB) | Element Count | Max Depth | Average Depth | F/B Size |
|---|---|---|---|---|---|
| DBLP [11] | 43.61 | 1103703 | 5 | 3.00 | 1158 |
| SwissProt [2] | 30.29 | 756329 | 6 | 4.39 | 21441 |
| XMark [22] | 5.34 | 78414 | 12 | 5.56 | 35558 |
| PSD [26] | 683.64 | 21305818 | 7 | 5.45 | 1944543 |
| Catalog [28] | 10.36 | 225194 | 8 | 5.65 | 235 |

**Table 1: Characteristics of experimental data sets.**

For our data sets, we chose DBLP [11], XMark [22], SwissProt [2], and the Protein Sequence Database [26]. These data sets have intrinsically different structures, ranging from the simplest (DBLP) to the most complicated (XMark) — Table 1 gives the salient aspects of each data set. For our update experiments, we used the catalog data set, generated by the XBench data generator [28].

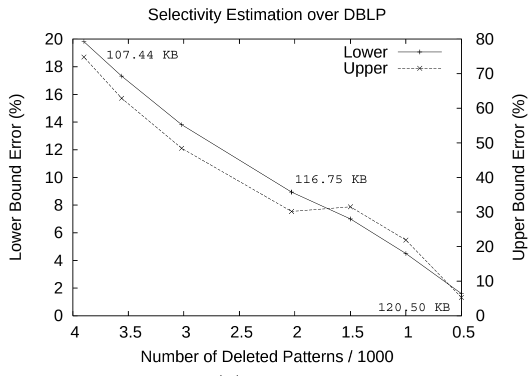
### 8.1 Evaluation of Estimation Quality

In our first experiment we test the quality of our selectivity estimation technique using randomly generated queries. We restrict our queries to branching path queries, having between $l$ and $u$ nodes (we chose the values $l = 3$ and $u = 5$ for our experiments). To generate queries, we make use of the full F/B-index of the data set in question, which contains the exact answers for all branching path queries.

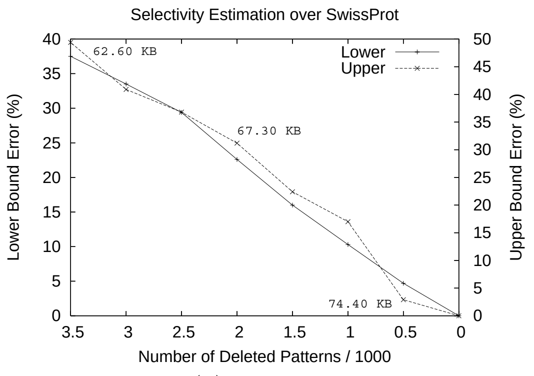
We generate each query as follows: first, we pick the number of nodes in the query by choosing an integer uniformly and at random in the range $[l, u]$. The *match node* of the query is selected at random over all nodes in the F/B index, with the probability of picking each node being its selectivity divided by $|\mathcal{D}|$. Thus, high selectivity nodes are favored. We then repeat the following process until we reach the desired number of nodes in the query: we pick an insertion point in the query at random, where the possible insertion points are at the root (i.e., inserting a new root node for the query), and at each node (i.e., inserting a new leaf node for the query). Once an insertion point is selected, we then randomly select a node from the relevant subset of the F/B index, biasing for high selectivity nodes.

We iterate the above procedure to generate a query workload of 100 queries. We constructed synopses using different values of the threshold parameter; for some values the corresponding sizes of the synopsis is shown in the graph. For each synopsis, we compare the selectivity estimates for each query with the exact selectivity. Our graphs report the *average relative error for both the lower and upper bound estimates.*

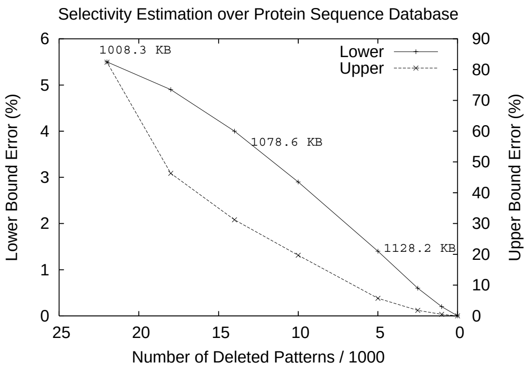
As can be seen, as the threshold parameter decreases the lower and upper bounds both correspondingly decrease. It is also clear that the upper bounds are less accurate than the lower bounds. One reason for this is due to our query workload, which consisted of twigs which make use of the `descendant-or-self` axis. This axis is particularly badly affected by the presence of $*$ nodes in the grammar, and hence the upper bounds tend to be higher. Nevertheless, the upper bounds are still well within a useful range, and the combination of an accurate lower bound and a slightly less accurate upper bound still give the query plan generator more information regarding the accuracy of the estimate than existing techniques.



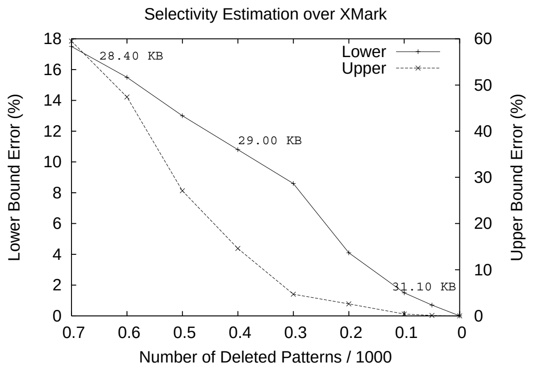
(a) DBLP



(b) SwissProt



(c) PSD



(d) XMark

**Figure 5: Relative error versus number of deleted patterns.**

## 8.2 Handling Updates

In this experiment we investigate the effect of updates on the size of the synopsis. Our updates were performed randomly on the catalog XML data set, in the following fashion:

1. An initial 80,000 node subset of the XML document is chosen at random to be the "seed" document.

2. Until the entire document is reconstructed, we randomly choose to either delete a node from the constructed tree, or insert a new subtree from the original document. The set of subtrees considered for insertion consists of all subtrees rooted at nodes of depth two in the original document, that are not yet included in the constructed document.

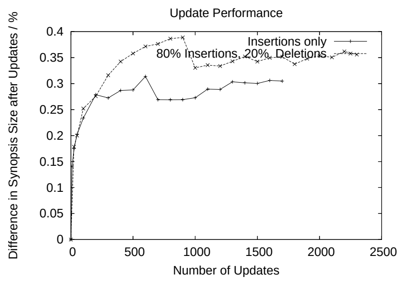
Figure 6(b) gives the results for two different runs of this experiment: one where no deletions are performed (1700 updates), and one where 20% of the operations are deletions (2300 updates). The graphs plot the relative size of the incrementally updated synopsis against the size of the synopsis that would be obtained if we recomputed the synopsis from scratch at that point. As can be seen, the space overhead imposed by updates remains relatively constant at about 40% of additional space. The initial spike in space usage is due to the fact that inserting or deleting nodes from the synopsis results in an initial "unrolling" of the grammar; however, due to the fact that XML documents are actually quite structured, after this initial increase in size it appears that there is little need to perform further unrolling.

We also note that if the updated synopsis becomes too large, its size can be reduced by running BPLEX on the underlying database again. This behavior can be seen in Figure 6(c), where we periodically, after each 400 updates, decompress and run BPLEX on the database again. As can be seen, the amount of space saved in this way is small and constant. This strengthens our belief that all updates can always be done on the grammar and that recomputation from the underlying database is not necessary. Note that for small documents it can happen that an updated synopsis becomes even smaller than the corresponding base synopsis; a similar effect can be seen in Fig. 6(c) where the recomputed synopsis seems to become larger than the updated one towards the end. This is due to the bottom-up search order of BPLEX and suggests that a randomized version of BPLEX will outperform the current implementation.

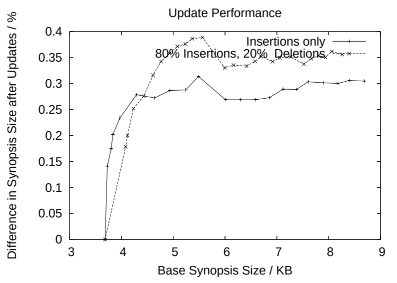
## 8.3 Discussion and Comparison

Our results demonstrate that our system can indeed handle a wide range of queries in a small space budget, and furthermore that the synopsis can efficiently be updated. It is clear that the lower bounds are more accurate than the upper bounds in our work, although the relative difference is dependent on the types of queries.

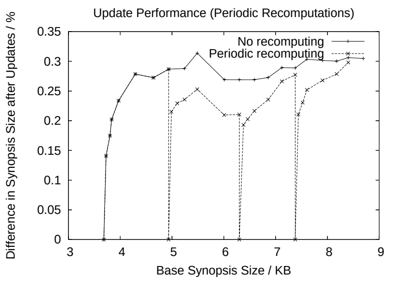
There is no related work which provides an equivalent set of features to our work. However, the most closely related works are the correlated sub-path trees (CSTs) of Chen et



(a) Updates



(b) Updates (against synopsis size)



(c) Updates (periodic recomputations)

**Figure 6: Update performance.**

al [6], the XSketch structural synopsis of Polyzotis et al [18], and StatiX [8]. The authors of [6] and [8] were kind enough to provide us with an implementation; unfortunately, we did not succeed in getting to run the implementations on our query workloads, even after substantial adaptations of queries and/or code. Therefore, we instead provide a direct comparison with the results presented in their papers.

Chen et al [6] reported errors of approximately 50%, using a synopsis size of 1% for DBLP and 5% for SwissProt. In contrast, as Figures 5(a) and 5(b) show, we obtain an error rate of less than 2% for lower bounds, and 10% for upper bounds, using a synopsis size of 120 KB (0.27%) for DBLP, and an error rate of about 2% for lower bounds, and 5% for upper bounds, using a synopsis size of about 62 KB (0.24%) for SwissProt.

We obtained an implementation of the TreeSketch estimation structure, which allows us to give a more direct comparison with the work of [17] (to our knowledge, this is the most competitive XML selectivity estimator currently available). We compared our work with this implementation using the XMark database, however, we had to slightly simplify the queries used to exclude order-sensitive axes and the descendant axis, which are not supported by TreeSketch (the latter only because it is not implemented). Our tests demonstrated that TreeSketch consistently gave relative errors in the range of 9-12% over the full range of synopsis sizes given in Figure 5(d). Therefore, while for smaller synopsis sizes TreeSketch clearly outperforms our approach, the two synopses converge in performance in the range of sizes given in the figure. It is worth keeping in mind that our synopsis is updateable and supports the full structural power of XPath, including the order-sensitive axes (not supported by TreeSketch). Moreover, even when using a non-optimized version of BPLEX, our synopsis can be computed extremely quickly: for a 5.4 MB XMark we need 8 seconds and for a 30 MB XMark approximately 30 seconds; as a comparison, the implementation of TreeSketch which we used takes 7 minutes for the 5.4 MB document and close to two hours for the 30 MB one.

It is difficult to determine the relative quality of StatiX and our work from the experimental results in their paper. It is clear that StatiX produces very accurate results, although as shown in our results with very small space it is possible to even produce exact results. Our work also handles a larger range of structural queries than StatiX, and is more amenable to updates. For example, the update strategy of IMAX occasionally requires a recomputation from the database, whereas our update strategy will never go back to the actual data.

## 9. CONCLUSIONS AND FUTURE WORK

In this paper, we have introduced a new selectivity estimation technique for structural XML queries that boasts several advantages over existing synopsis structures: it supports all thirteen XPath axes, whilst also being amenable to efficient updates. Instead of returning educated guesses, as many other techniques do, we instead return a range within which the selectivity is guaranteed to lie. We believe that this is particularly useful for query optimizers, as it allows them to determine the relative confidence of two selectivity estimates. Our experimental results have demonstrated that our approach, despite its additional features, is competitive with existing techniques, in both accuracy and space. Our synopsis is a holographic representation of the XML document tree and might be useful for other applications besides selectivity estimation.

In the future, we plan to extend our techniques to handle XML data values as well as structural queries. A possible way of doing this is to keep seperate from the tree structure a synopsis for data values which is built using conventional techniques. The challenge then is to have an efficient way to fetch at a leaf node of our synopsis the corresponding (estimatation of the) data value. Another possibility of handling data values is to store them symbolically as part of the tree structure, and to apply our compression techniques directly on the tree. This approach is promising. Consider, for instance, string values stored as monadic trees; for such trees our technique achieves high compression as it corresponds to Lempel-Ziv-like string compression. Unlike before, only lengths have to be stored when pruning, and, the string after a pruning can still be used in the selectivity estimation of the query.

## Acknowledgment

The authors gratefully thank Neoklis Polyzotis for providing us with an implementation of his TreeSketch estimation framework.

## 10. REFERENCES

[1] A. Aboulnaga, A. R. Alameldeen, and J. F. Naughton. Estimating the selectivity of XML path expressions for internet scale applications. In *VLDB*, pages 591–600, 2001.

[2] A. Bairoch et al. The universal protein resource (UniProt). *Nucleic Acids*, 33:154–159, 2005.

[3] M. A. Bender et al. Two simplified algorithms for maintaining order in a list. In *ESA*, pages 152–164, 2002.

[4] P. Buneman, M. Grohe, and C. Koch. Path queries on compressed XML. In *VLDB*, pages 141–152, 2003.

[5] G. Busatto, M. Lohrey, and S. Maneth. Efficient memory representation of XML documents. In *DBPL*, pages 199–216, 2005.

[6] Z. Chen et al. Counting twig matches in a tree. In *ICDE*, pages 595–604, 2001.

[7] J. Clark and S. DeRose. XML path language (XPath) version 1.0. http://www.w3.org/TR/xpath.

[8] J. Freire, J. R. Haritsa, M. Ramanath, P. Roy, and J. Siméon. StatiX: making XML count. In *SIGMOD*, pages 181–191, 2002.

[9] G. Gottlob, C. Koch, and R. Pichler. Efficient algorithms for processing XPath queries. *ACM ToDS*, 30(2):444–491, 2005.

[10] J. Lamping. An algorithm for optimal lambda calculus reductions. In *POPL*, pages 16–30, 1990.

[11] M. Ley. Digital bibliography and library project. http://dblp.uni-trier.de/.

[12] L. Lim, M. Wang, S. Padmanabhan, J. Scott Vitter, and R. Parr. XPathLearner: An on-line self-tuning markov histogram for XML path selectivity estimation. In *VLDB*, pages 442–453, 2002.

[13] M. Lohrey and S. Maneth. Tree automata and XPath on compressed trees. In *CIAA*, pages 225–237, 2005.

[14] D. Olteanu, H. Meuss, T. Furche, and F. Bry. XPath: Looking forward. In *EDBT Workshops*, pages 109–127, 2002.

[15] N. Polyzotis and M. N. Garofalakis. Statistical synopses for graph-structured XML databases. In *SIGMOD*, pages 358–369, 2002.

[16] N. Polyzotis and M. N. Garofalakis. Structure and value synopses for XML data graphs. In *VLDB*, pages 466–477, 2002.

[17] N. Polyzotis, M. N. Garofalakis, and Y. E. Ioannidis. Approximate XML query answers. In *SIGMOD*, pages 263–274, 2004.

[18] N. Polyzotis, M. N. Garofalakis, and Y. E. Ioannidis. Selectivity estimation for XML twigs. In *ICDE*, pages 264–275, 2004.

[19] M. Ramanath, L. Zhang, J. Freire, and J. R. Haritsa. IMAX: The big picture of dynamic XML statistics. In *ICDE*, pages 273–284, 2005.

[20] C. Sartiani. A framework for estimating XML query cardinality. In *WebDB*, pages 43–48, 2003.

[21] C. Sartiani. A general framework for estimating XML query cardinality. In *DBPL*, pages 257–277, 2003.

[22] A. Schmidt et al. XMark: A benchmark for XML data management. In *VLDB*, pages 974–985, 2002.

[23] H. Seidl. Finite tree automata with cost functions. *TCS*, 126:113–142, 1994.

[24] W. Wang, H. Jiang, H. Lu, and J. Xu Yu. Containment join size estimation: Models and methods. In *SIGMOD*, pages 145–156, 2003.

[25] W. Wang, H. Jiang, H. Lu, and J. Xu Yu. Bloom histogram: Path selectivity estimation for XML data with updates. In *VLDB*, 2004.

[26] C. H. Wu et al. The protein information resource. *Nucleic Acids Research*, 31:345–347, 2003.

[27] Y. Wu, J. M. Patel, and H. V. Jagadish. Estimating answer sizes for XML queries. In *EDBT*, pages 590–608, 2002.

[28] B. Bin Yao, M. Tamer Özsu, and N. Khandelwal. XBench benchmark and performance testing of XML DBMSs. In *ICDE*, pages 621–633, 2004.