Structural Selectivity Estimation for XML Documents

Damien K. Fisher and Sebastian Maneth
Sydney Research Lab, National ICT Australia*
and School of Computer Science and Engineering, UNSW
Sydney, Australia
{ damien.fisher, 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. 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, 12, 6, 9, 23, 20, 15, 16, 17, 18, 21]. All previous work suffers from some combination of the following problems:

Expensive construction: For many techniques, 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,

*National ICT Australia is funded through the Australian Government's *Backing Australia's Ability* initiative, in part through the Australian Research Council

1-4244-0803-2/07/$20.00 ©2007 IEEE.

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 [21], no previous technique 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 approaches:

- 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 less than for other synopses.
- Our synopsis can give selectivity estimates for all structural XPath queries, including the 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 a complete solution to the problem of selectivity estimation for structural queries.

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 [9] 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.

Research upon the estimation of result sizes for the structural join operator, such as [23, 20], is also relevant to selectivity estimation for path expressions, because selectivity estimates for paths of the form $//p_1//p_2$ are given. Unfortunately, these results cannot be easily generalized.

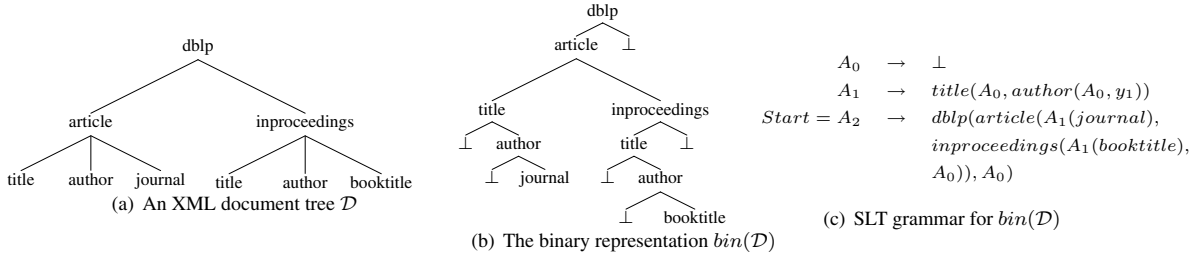
Polyzotis and Garofalakis [15-18] develop a general framework, XSKETCH, which provides good estimates for twig queries using graph synopses. The primary shortcoming of this method is that the construction process for the synopsis is expensive, and also relies on generating a set of test queries upon 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 [21], 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.

The rest of the paper is organized as follows. After a formal definition of the problem in Section 2, we introduce our synopsis structure, *straight-line tree grammars*, in Section 3. In Section 4, we show how to use tree automata over these grammars as an effective means of computing selectivity. We then discuss how to efficiently update our synopsis in Section 5, as well as how to efficiently store it in Section 6. Section 7 presents an experimental analysis of our techniques.

2 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



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

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

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

straightforward fashion). By Σ we denote the alphabet of elements present in $\mathcal{D}$.

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. Figure 1 shows an XML document in its unranked representation, and also gives the corresponding ranked representation, $bin(\mathcal{D})$. We use $\perp$ to denote the empty tree, and write $V_{\mathcal{D}}$ for the vertices of the document (in the ranked representation), and $\lambda : V_{\mathcal{D}} \rightarrow \Sigma$ for the mapping from vertices of the document to their labels.

Queries Core XPath [10] is a powerful fragment of XPath that can be seen as the structural portion of XPath. It consists of queries satisfying the following grammar:

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

In this grammar, χ 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 $t = *$). For ease of presentation we only consider conjunction in this paper (our results are easily generalized to handle other Boolean functions).

We represent a core XPath 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 a query is a tree, each node q in the query has at most one parent — therefore, for convenience we write $\lambda_E(q) = \lambda_E(\langle \textsc{Parent}(q), q \rangle)$. One of the vertices of Q, $m_Q \in V_Q$, is the *match node* (cf. the boxed node in Fig.2(a)). The semantics of an XPath query is well-known [7], and so we only briefly summarize it here. An *embedding* of a query Q in a document $\mathcal{D}$ is a tree homomorphism $h : V_Q \rightarrow V_{\mathcal{D}}$ such that, for every node v of Q, $h(v)$ has the same label as v (or $\lambda_V(v) = *$), and each edge e of Q is mapped into two nodes that satisfy the constraints specified by $\lambda_E(e)$. For instance, if $\lambda_E(\langle v_1, v_2 \rangle) = \texttt{child}$, 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: $Q(\mathcal{D}) = \{h(m_Q) \mid \text{for all embeddings } h \text{ of } Q \text{ in } \mathcal{D}\}$. 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: since backward axes can be eliminated [14] we only use forward axes. Additionally, we rewrite `descendant` 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.

3 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 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 is later used to overestimate 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 6), and (2) they can be queried in a direct and natural way without prior decompression [13]. In particular, it is shown in Section 4 how to translate XPath queries into certain tree automata which can be executed on SLT grammars.

Tree Compression using SLT Grammars Most XML documents are highly repetitive: the same tags ap-

pear 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. A tree pattern is a tree with "holes" and can be represented by a tree by filling formal parameters $y_1, y_2, \ldots$ into the holes. Thus, we represent $c(d($ as $c(d(y_1), y_2)$. A tree can now be represented by an array $[A_1, .., A_n]$ of tree patterns where each tree pattern A_i may refer to preceding tree patterns (i.e., to A_j with $j < i$). For example, the tree $c(d(e(u)), c(d(f), c(d(a), a)))$ is represented by the array $[A_1 : c(d(y_1), y_2), A_2 : A_1(e(u), A_1(f, A_1(a, a)))]$. To read back the tree from the array ("unfolding") we start with the last entry A_n and then recursively replace in it references A_i by the corresponding tree patterns. Such arrays are also called *Straight-Line context-free Tree grammars (SLT grammars)*, an entry $A_i : t$ is called a "rule", and A_i is called a "nonterminal". We use the BPLEX algorithm [5] which approximates in linear time a small SLT grammar for a given tree. Since we work on binary tree representations, the tree above is represented by the following grammar $H = [A_0 : \bot, A_1 : a, A_2 : c(d(y_1, y_2), A_0), A_3 : A_2(e(u, A_0), A_2(f, A_2(A_1, A_1)))]$.

Lossy Compression Consider an XML document tree $\mathcal{D}$ and an SLT grammar G representing $bin(\mathcal{D})$. We want to reduce G's size so that the result can be used for selectivity estimation. We do this by deleting tree patterns, starting with the least frequently used ones. Given a number κ, the *(κ-) lossy grammar (for G)* is obtained from G by deleting the κ least frequently used patterns. The frequency of each pattern A_i can be determined by multiplying corresponding numbers of references in one top-down (right-to-left in the array) run through the grammar G. If the ith entry $A_i : t$ is deleted, then we replace each reference A_i (with subtrees $t_1, \ldots, t_k$) in the remaining grammar by the following tree

$$\begin{cases} *(t_1, \ldots, t_k, h, s) & \text{if } k = 0 \text{ or } t\text{'s right-most leaf is } y_k \\ *(t_1, \ldots, t_k, \bot, h, s) & \text{otherwise} \end{cases}$$

where h and s are the height and size, respectively, of the unranked tree corresponding to the unfolding of A_i. The numbers h and s are stored to later over-estimate the selectivity. As an example, consider the grammar H shown at the end of the previous subsection. The 1-lossy grammar for H is $[A_0 : \bot, A_2 : c(d(y_1, y_2), A_0), A_3 : A_2(e(u, A_0), A_2(f, A_2(*(1, 1), *(1, 1))))]$. Note that if we were deleting A_2 from the grammar, then we would have to replace $A_2(t_1, t_2)$ by $*(t_1, t_2, \bot, 2, 2)$ because the right-most leaf of the pattern A_2 is $A_0 \neq y_2$.

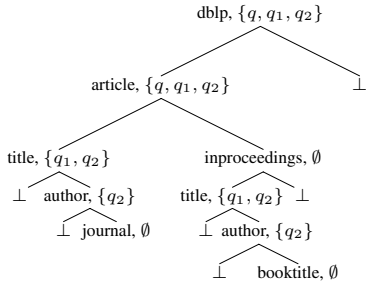
Later we will evaluate XPath queries on lossy grammars. For this it is important to understand the unranked semantics of a tree $*(t_1, t_2, t_3, h, s)$: it represents any sequence $\sigma = g_1, g_2, \ldots, g_n$ of trees such that: (1) the sequence $g_1, \ldots, g_{n-1}$ has subtrees t_1 and t_2 (2) the last tree g_n equals t_3 (3) the height of the sequence σ is h (4) the size of the sequence σ is s.

4 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 method is generalized to work over SLT grammars. The final step is to handle lossy SLT grammars which contain $*$s in rules. We use examples to describe the construction of the tree automata; a more detailed formalization can be found in [8].

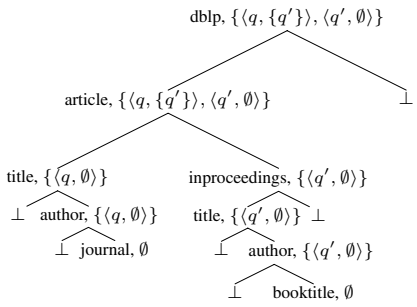
A (deterministic) *tree automaton* over ranked tree encodings consists of a finite set P of *states*, a set $F \subseteq P$ of *final states*, and of a transition function $\delta : P \times P \times \Sigma \to P$. The automaton is run on a tree in a bottom-up fashion: the empty trees ($\bot$) which appear at the leaves are assigned the empty state, $\emptyset$. We then move upwards and assign to a node with label a and children that have been assigned states p_1 and p_2, the state $\delta(p_1, p_2, a)$. The tree is accepted by the automaton if the root node is assigned a final state.

Converting Queries to Tree Automata The translation of a core XPath query into a tree automaton is based on 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 sub-queries: q itself, $q_1 =$ `//title`, and $q_2 =$ `//author`. Working in a bottom-up fashion on the document tree in Figure 1(b), 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 q matches the document if both q_1 and q_2 match the left child, and if the label of the node is `article`. 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 a query that selects `author` nodes which have a `title` node in their `following` axis. 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 `article` 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 Figure 1(b) results in:



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. 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 automata a set of counters. Our annotated states $\langle p, C\rangle$, consist of a normal state, $p \in 2^{Q\times 2^Q}$, and an array of counters, so that $C[\langle q, U\rangle]$ is the counter for each $\langle q, U\rangle \in p$. We will assume that $C[\langle q, U\rangle] = 0$ if $\langle q, U\rangle \notin p$. Each counter represents the number of nodes matching the corresponding sub-query, that have not already been matched by that sub-query'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.

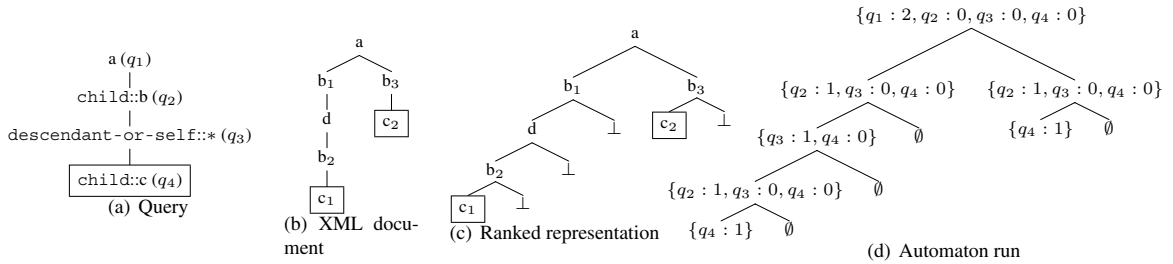
There are two issues that are worth mentioning. When we match a new query node, the match count for that query node is clearly the sum of the counts of its children. However, once we have copied over the children counts, we must zero them out as well. This is to prevent double-counting, which occurs 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 subquery q_2 in Figure 2(a). 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 sub-query — however, its child, the sub-query q_3, *is* a matching sub-query. Therefore, when removing q_2 from the set of matching sub-queries, we must transfer its count of matching nodes back to q_3.

Tree Automata over SLT Grammars Up till this point we have considered tree automaton 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 $A_i : t$ and t uses parameters $y_1, \ldots, y_k$, 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 σ_i, we will need to make calls to the functions σ_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; 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, $f_i(p_1, \ldots p_k) \rightarrow \mathbb{Z}$, can be determined by a natural extension of our tree automaton of before. Hence it is easy to extend σ_i to also compute f_i. When we come across a non-terminal $A_i(t_1, \ldots, t_k)$ in the



(a) Query (b) XML document (c) Ranked representation (d) Automaton run

right hand side of a rule, we can compute its state by first recursively determining the states $p_1, \ldots p_k$ corresponding to the input parameters, and using these and σ_i to determine the corresponding state (and selectivity counts) for the nonterminal.

Determining complexity of this algorithm is straightforward. If every rule has at most k parameters, then selectivity counting over a straight-line grammar G 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 obtain as time complexity $O((|Q|/b)^{bk}2^{mk}|G|)$.

In practice, BPLEX returns very small grammars even with a very low value for k, and so we can ignore this dependency. Also, we suspect that in practice the branching factors of queries is also lower, and 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 σ_i, but instead can lazily compute only those values we need: we find that in practice only a small number of combinations of states are seen, and so this algorithm runs quickly.

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. 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.

Estimating upper bounds is straightforward conceptually, but the details are quite involved. 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 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 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.

5 Incremental Updates

In this section, we present an effective update algorithm for lossless SLT grammars. It can be applied to a lossy grammar by keeping a lossless grammar on-disk, and referencing it only for those updates that occur at a star (deleted) position. We consider three update operations: *first_child*(p, t), *next_sibling*(p, t), and *delete*(p), where p is the path to a node v, in Dewey notation, and t is a tree. They insert t as first child of v, as the next sibling of v, or delete the subtree rooted at v, respectively.

An update is realized by expanding in the start rule of the grammar all references A_i that occur on the path to v.

Once only terminal symbols are on the path to v, the update can be executed. The resulting tree is now run through BPLEX again (using the existing grammar when looking for patterns). This will shrink the tree (almost) back to its original size plus the size of possibly inserted (compressed) tree. These changes are *incremental* in the sense that they are local: only the start rule is changed, and (possibly) new rules are added. 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. Clearly, only linear time is needed for performing an update.

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 of before: $[A_0 : \perp, A_1 : a, A_2 : c(d(y_1, y_2), A_0), A_3 : A_2(e(u, A_0), A_2(f, A_2(A_1, A_1)))]$ and the update operation *first_child* 1.2.1 $e(u)$. We rewrite the start right-hand side until no nonterminals are on the path from the root to the node 1.2.1. 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. It discovers a new pattern $e(u, y_1)$ that appears twice and therefore adds the new nonterminal A_3. The final grammar after update is: $[A_0 : \perp, A_1 : a, A_2 : c(d(y_1, y_2), A_0), A_3 : e(u, y_1), A_4 : A_2(A_3(A_0), A_2(A_3(f), A_2(A_1, A_1)))]$.

6 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 one to access arbitrary nodes in the tree 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.

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

Symbol Table

$* \to 0$ $\quad a \to 2$ $\quad A_0 \to 5$

$y_i \to 1$ $\quad b \to 3$ $\quad A_1 \to 6$

$\quad c \to 4$

$*$ Statistics

0 $\quad h = 5, s = 10$

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 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 4).

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 replace patterns that occur more than once, each $\langle h, s\rangle$ occurs 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.

Figure 3 shows how a rule is encoded: if there are m parameters, then we start with m 1-bits followed by a 0-bit. After this, symbols are encoded in their order of appearance (pre-order of the tree), using the symbol table. Since there are $*$, y_i, A_0, and symbols in Σ, we need $\log(|\Sigma| + i + 3)$ bits for each symbol. Additionally, since the number of arguments of a $*$-node are not known a priori, we prefix the encoding of each direct subtree by a single 1-bit, followed by a single 0-bit and the symbol table entry for the $\langle h, s\rangle$ pair. 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).

7 Experiments

In this section we give an empirical evaluation of our system. It is quite difficult to test selectivity estimation since practical XML query workloads are not yet known, and hence we used synthetic queries. 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 reported here assume that the packed representation was used (sizes are plotted at data points in Fig. 4). The BPLEX algorithm was run with maximal rank 5, maximal right-hand side size 20, and window size 1000, cf. [5] for more details on these parameters.

For our data sets, we chose DBLP [11], XMark [19], SwissProt [2], and the Protein Sequence Database [22]. These data sets have intrinsically different structures, ranging from the simplest (DBLP) to the most complicated (XMark) The following table gives the salient aspects.

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 [19]	5.3	78414	12	5.56	35558
PSD [22]	683.64	21305818	7	5.45	1944543
Catalog [24]	10.36	225194	8	5.65	235

Evaluation of Estimation Quality In our first experiment we test randomly generated queries. We restrict our queries to branching path queries, using only the `descendant-or-self` axis, and having between l and u nodes ($l = 3$ and $u = 5$ for our experiments). Queries are generated using the full F/B-index. 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, where the probability for picking a node is its selectivity divided by $|\mathcal{D}|$. Thus, high selectivity nodes are favored. We then add the required nodes by randomly selecting from the relevant subset of the F/B index, biasing for high selectivity nodes. We generate a query workload of 100 queries and compare, for each synopsis, 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. The amount of memory needed to hold a synopsis is shown inside the graphs, for some points.

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 purely of twigs using 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.

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.

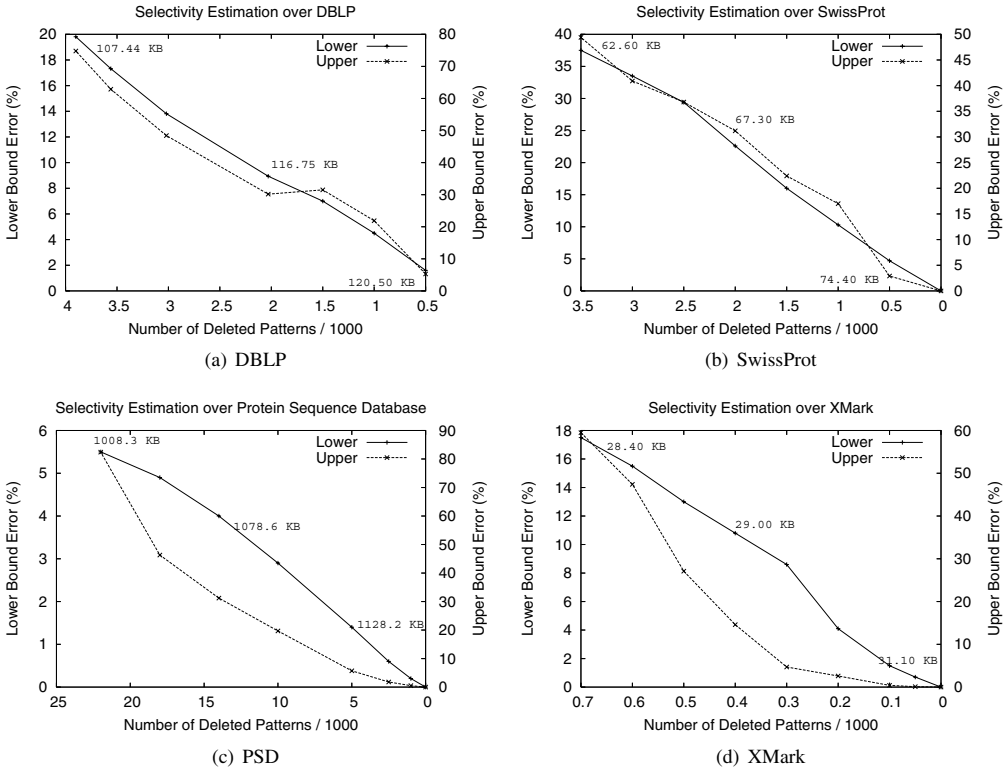
(2)Until the entire document is reconstructed, we randomly choose to either delete a node from the constructed tree, or insert a new subtree from the original document. The set of subtrees considered for insertion consists of all subtrees rooted at nodes of depth two in the original document, that are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

Figure 5(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 5(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 believe that all updates can always be done on the grammar and that recomputation from the underlying database is not necessary.

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 significantly more accurate than the upper bounds in our work, although the relative difference is dependent on the types of queries. This suggests that the upper bound should be used as a measure of confidence in the result.

There is no related work which provides an equivalent set of features to our work. However, the two most closely related works are the correlated sub-path trees (CSTs) of Chen et al [6] and the TREESKETCH structural synopsis of



(a) DBLP (b) SwissProt

(c) PSD (d) XMark

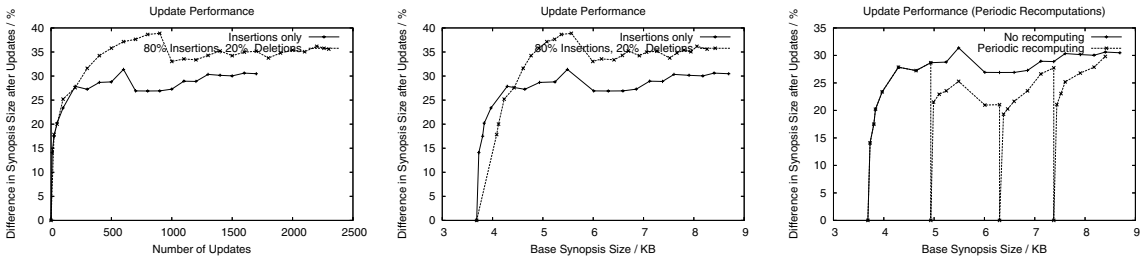
Polyzotis et al [17].

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 4(a) and 4(b) show, we obtain an error rate of less than 10% using a synopsis size of 120 KB (0.27%) for DBLP, and an error rate of about 40% using a synopsis size of about 62 KB (0.2%) 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 4(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.

8 Conclusions and Future Work

In this paper, we have introduced a new selectivity estimation technique for structural XML queries that has 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, it instead returns 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



(a) Updates (b) Updates (against synopsis size) (c) Updates (periodic recomputations)

with existing techniques, in both accuracy and space.

In the future, we plan on extending our techniques to handle XML data values as well as structural queries. A natural extension to this is to consider data values symbolically (i.e., as monadic subtrees of the document tree), and to apply our compression techniques directly to this tree. While we believe this approach is promising, the large number of unique data values poses additional challenges to selectivity estimation.

Acknowledgment We are grateful to Neoklis Polyzotis for providing us with an implementation of his TREESKETCH estimation framework.

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] D.K. Fisher and S. Maneth. Structural selectivity estimation for XML documents. Technical Report PA006152, National ICT Australia, 2006.

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

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

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

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

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

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

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

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