A Decomposition-Based Probabilistic Framework for Estimating the Selectivity of XML Twig Queries

Chao Wang, Ruoming Jin, Srinivasan Parthasarathy

Department of Computer Science and Engineering, The Ohio State University
Contact: {wachao, jinr, srini}@cse.ohio-state.edu

Abstract

In this paper we present a novel approach for estimating the selectivity of XML twig queries. Such a technique is useful for approximate query answering as well as for determining an optimal query plan, based on said estimates, for complex queries. Our approach relies on summary structure that contains occurrence statistics of small twigs. We then present a novel probabilistic approach for decomposing larger twig queries into smaller ones. We then show how in conjunction with the summary information it can be used to estimate the selectivity of the larger query. We present and evaluate two approaches for decomposition and compare this work against a state-of-the-art selectivity estimation approach on synthetic and real datasets. Quantitatively, our results show that the new approach is much more efficient in terms of the time it takes to construct the summary and estimate the selectivity of a twig query. Qualitatively, the new approach is more accurate on most datasets.

1 Introduction

XML is gaining acceptance as the standard for data representation and exchange over the World Wide Web. However, for wide spread deployment and use it is becoming increasingly clear that the design of an efficient high level querying mechanism is necessary. Since XML documents may be represented as a rooted and labeled tree, this necessity has led to the development of tree-based (twigs) querying mechanisms. Twig queries describe a complex traversal of the document graph and retrieve document elements through an intertwined (i.e., joint) evaluation of multiple path expressions.

Given the importance of twig queries as a basic selection mechanism in XML[15, 14, 3], efficient support for accurately estimating the selectivity of twig queries is crucial for query optimization of complex queries. This is analogous to selectivity estimation in relational databases [4, 6, 7, 11]. Accurate selectivity estimation is also desirable in interactive settings and for approximate querying. For instance, an end-user can interactively refine her query if she knows that the current query will result in an overwhelming result set. Similarly, the estimated value can be returned as an approximate answer to aggregate queries using the COUNT primitive.

The early work in this area has focused on determining the selectivity of path expressions (a special case of twig queries) [10, 9, 1, 17, 16, 8]. The Lore system [10] adopts a Markov model based approach for this purpose. Markov table[1], improves on the Lore system through the use of intelligent pruning and aggregation to reduce the space requirements.Recently, Lim and Wang proposed XPathLearner [9], an on-line tunable Markov table method which has been shown to be effective for path expression selectivity. A key limitation of these methods is that they do not adapt to twig queries well since path correlations are not accounted for.

More recently researchers have focused on selectivity estimation for twig queries [5, 3, 13, 14, 15]. Examples include Correlated Sub-Trees [3], XSketches [13, 15], and TreeSketches [14]. Among these it has been shown that TreeSketches is the most accurate and efficient method presented to date [14]. TreeSketches [14], a successor of XSketches, clusters the similar fragments of XML data together to generate its synopsis. The granularity of the clustering depends on the memory budget.

To estimate the selectivity of XML twig queries,

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

Proceedings of the 31st VLDB Conference, Trondheim, Norway, 2005

all the above approaches as well as the approach presented in this article define a summary data structure that houses important statistics about the data from which the selectivity may be estimated. Important issues at hand include: the quality of estimation from the given summary; time to construct the summary; and finally the time to estimate the selectivity of queries from the summary. To address these issues we present a new approach to selectivity estimation. The key contributions of our approach are highlighted below.

First, we present a probabilistic framework under which the selectivity of a query (represented as a rooted tree) can be estimated from its subtrees. We present and evaluate two different strategies for decomposing the query into subtrees. These subtrees can then be used to arrive at a selectivity estimate. We present a theoretical basis for this approach. We also show that our decomposition framework subsumes the Markov model based XML path selectivity estimation as a special case.

Second, to summarize an XML dataset we leverage the use of frequent tree mining. A dynamically determined subset[1] of all the occurred subtrees up to a certain size (number of nodes), coupled with associated occurrence statistics, forms the basis of our summary structure. More specifically the dynamic subset we store is based on the notion of *(non)-derivable* occurred patterns. We also rely on fast searching mechanisms to locate the subtrees of a given twig query within our summary structure.

Third, we conducted an extensive experimental study to examine the benefits of our approach and compare it against TreeSketches[2]. Empirical results show that our approach takes several orders of magnitude less time to construct the summary, and is one to two orders of magnitude faster when computing the selectivity estimates. In our qualitative assessment we also find that our approach compares favorably with TreeSketches. We also offer a detailed explanation as to why the new approach(labeled *TreeLattice*) outperforms TreeSketches [14]).

2 Problem Definition and Related Work

2.1 Problem Definition

An XML document can be structurally modeled as a tree (if one ignores IDREFs) where each node is typically associated with a tag or a value. In practice, values are almost always associated with leaf nodes and tags with interior nodes. Similar to prior work by Polyzotis and Garofalakis [12], in this paper we do not model value elements.

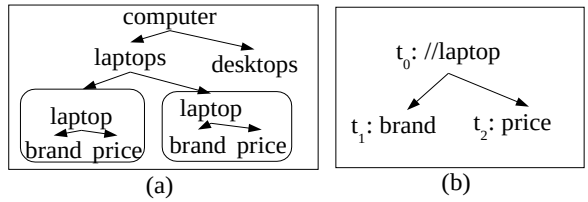


Figure 1. (a)A sample XML data tree; (b)A sample twig query

More formally, let Σ be an alphabet, let Σ^* be the set of strings of finite length on Σ, and $II \subset \Sigma^*$ be a small set of strings. Let T (representing an XML document) be a large rooted node-labeled tree $T = (V_T, E_T)$ where non-leaf nodes are labeled with strings from II (element tags and attribute names), and leaf nodes are labeled with strings from Σ^*. Figure 1(a) shows a sample xml document containing online auction information.

A twig query T_Q is defined as a node-labeled tree $T_Q(V_Q, E_Q)$, where each node $t_i \in V_Q$ is labeled with a path expression P_i. At an abstract level, each node t_i corresponds to a subset of elements, while path P_i describes the structural relationship that must be satisfied between elements in t_i and elements in its parent node.

We next define the notion of a twig match given by Chen *et al.* [3].

Definition 1 *A match of a twig query $T_Q = (V_Q, E_Q)$ in a node-labeled data tree $T = (V_T, E_T)$ is defined by an $1-1$ mapping: $f : V_Q \mapsto V_T$ such that if $f(u) = v$ for $u \in V_Q$ and $v \in V_T$, then (i) Label(u) = Label(v) and (ii) if $(u, u') \in E_Q$, then $(f(u), f(u')) \in E_T$.*

The selectivity $\sigma(T_Q)$ of twig query T_Q is defined as the number of matches of T_Q in the data tree. Figure 1(b) shows a sample twig query and the encircled subtrees in Figure 1(a) show its two matches in the sample XML data tree. Our objective is to accurately estimate the selectivity of an XML twig query Q in as efficient a manner as possible given constraints in space (summary storage) and time (summary construction and estimation time).

2.2 Related Work

Chen *et al.* [3] are among the first to study the problem of estimating twig counts. They propose the Correlated Sub-path Tree(CST) method for estimating the selectivity for XML twig queries. A CST is a suffix tree based data structure to store all the paths up to

[1]Due to storage costs, the complete lattice (all frequent patterns) cannot be held in memory, we can only store a portion of it which is data dependent and dynamic.

[2]We are grateful to Neoklis Polyzotis for providing us with the TreeSketches executable and also for helping us tune the approach for a fair comparison.

certain length. To estimate the selectivity of a given twig query, this approach needs to decompose the twig into a set of paths stored in the CST. Note that even CST approach and our TreeLattice approach both depend on decomposing a large twig into the basic twigs, two approaches are quite different in several perspectives. First, our approach utilizes the *subtrees* instead of paths as the summary of an XML document to estimate a twig query. Our results have shown the subtrees capture the structure of an XML document very effectively. In contrast, CST has to store additional information associated with each path, denoted as the *set hashing signature*, to capture the correlation among paths, in order to perform the selectivity estimation. Also, our approach is essentially an generalization of the Markov model based approach for XML path selectivity estimation(Subsection 3.4). Note that when dealing with XML path selectivity, the Markov property based approach has been shown to be more effective than the CST-based approach [3].

XSketch [12] exploits localized graph stability in a graph-synopsis model to approximate path and branching distribution in an XML data graph. Its successor, XSketches [13], integrates support for value constraints as well, by using a multidimensional synopsis to capture value correlations. They augment XSketch model [15] with new distribution information to estimate the selectivity of XML twig queries. Also, they show that XSketches performs better than CST and yields estimates with significantly lower estimation error.

TreeSketches [14], a successor of XSketches, clusters the similar fragments of XML data together to generate its synopsis. The granularity of the clustering depends on the memory budget. Also, it outperforms its predecessors, in terms of both accuracy and construction time.

A particular case of the twig query is the XML path query. The wide use of the XML path queries has motivated many researches on estimating the selectivity of XML path queries. Lore system [10] is one of the early work in this direction. It stores statistics of all distinct paths up to length m, where m is a tunable parameter. Selectivity of paths longer than m are estimated assuming the Markov property. Aboulnaga *et al.* [1], extends the idea used by Lore system in their Markov table method. The Markov table method consists of a set of pruning and aggregation techniques on the statistics used in the Lore system and is therefore an improvement over the method used in the Lore system because it reduces the space requirement. Aboulnaga *et al.* [1], also proposes a tree-based method known as path tree, for estimating the selectivity of XML paths without data values. A path tree is a summarized form of the XML data tree. Compared with the Markov table method, this approach is inferior in terms of estimation accuracy on real data sets [1].

XPathLearner [9], is an on-line self-tuning Markov table based approach to estimate the selectivity of XML paths. The statistics of the data is collected in an on-line fashion, thus is workload aware. Our approach by design is also incremental in nature and can maintain summaries on-line although we do not evaluate this aspect in this paper. We note that our approach is provably a generalization of these Markov model based approach for more complex twig queries. Recently, Wang *et al.* [16] propose the use of Bloom Histograms to estimate XML path selectivity. It is the first approach that gives a theoretical bound on the estimation error. However, it does not handle twig queries.

3 An Estimation Framework based on Twig Decomposition

Similar to the previous Markov Table approach to the XML path selectivity estimation problem, we use the counting information of small twigs as the summary of the original XML data. Since we rely on the lattice-based framework for collecting such information, we name this structure *lattice summary*. The lattice summary consisting of twigs of size m or less is denoted as L_m or m-lattice. For twig selectivity estimation, if the twig is small and it's in the lattice summary, then we just need to retrieve its count from the summary directly and use this as the estimate. On the other hand, if the twig is large, and we cannot obtain its count directly from the lattice summary, then we have to come up with novel approaches for this.

The basic question we seek to answer in this section is given the lattice summary L_m, how can we estimate the selectivity of a twig query of size n where $n > m$ accurately. The details of the lattice summary will be described in the next section.

Before we answer this question we will first detail our solution to the simpler problem of estimating the selectivity of a twig of size n from two twigs of size $n-1$ with an $n-2$ sized common subtree under the assumption of conditional independence.

3.1 Augmenting Twigs

Suppose we have two basic twigs T_1 and T_2, and they differ by only one edge (Figure 2(a)). If T is their common part, then we can express T_1 as $T \cup \{e_1\}$ and $T_2 = T \cup \{e_2\}$, where e_1 and e_2 are the two *distinct* edges. The edges are distinct in that they either attach to different nodes of T, or the two additional nodes x and y introduced by these two edges are different. For the sake of expository simplicity, we assume that for a given parent all children are distinct within a twig query. The two twigs can be augmented together to generate a larger twig, denoted as

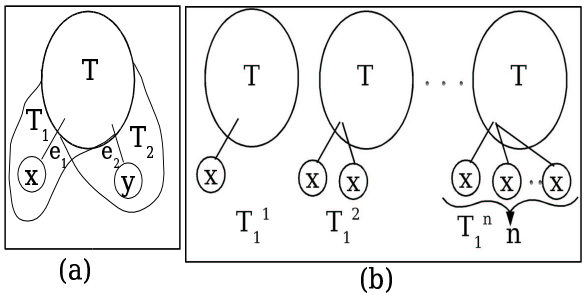


Figure 2. (a) Augmented twigs $T_1 \cup T_2$; (b) Growing T_1 from T

$T_1 \cup T_2 = T \cup \{e_1\} \cup \{e_2\}$. Assuming the counts of T_1 ,T_2, and their common part T are available, denoted as $\sigma(T_1)$, $\sigma(T_2)$, and $\sigma(T)$ respectively, we are interested in estimating the count of the augmented twig, $T_1 \cup T_2$, based on these information.

A complication we have here is that in the XML document the occurrence of T could be coupled with one or more instances of edge e_1 as shown in Figure 2(b). Let T_1^i denote the occurrence of T with i edges of type e_1. Then it is easy to see that the selectivity of T_1 is given by the decomposition formula[3]:

$$\sigma(T_1) = \sigma(T_1^1) + 2 \times \sigma(T_1^2) + \cdots n \times \sigma(T_1^n)$$

and similarly the selectivity of T_2 is given by:

$$\sigma(T_2) = \sigma(T_2^1) + 2 \times \sigma(T_2^2) + \cdots m \times \sigma(T_2^m)$$

In order to derive our formula for estimating the augmented twig $T_1 \cup T_2$, we assume that the event of growing T1 from T is conditionally independent from the event of growing T2 from T. More formally we have:

$$Pr(T_1^i \cup T_2^j | T) = Pr(T_1^i | T) \times Pr(T_2^j | T)$$

where:

$$Pr(T_1^i | T) = \sigma(T_1^i)/\sigma(T)$$

and:

$$Pr(T_2^i | T) = \sigma(T_2^i)/\sigma(T)$$

Theorem 1 *Given two non-trivial rooted and labeled trees T_1 and T_2, which differ by only one edge, let T be the common part between T_1 and T_2, then the expected count of $T_1 \cup T_2$ can be estimated by $\sigma(T_1) \times \sigma(T_2)/\sigma(T)$.*

Proof: Given the tree growing independence assumption, we can treat the counts of $T_1 \cup T_2$ as a random variable. The expected counts of this random variable, $E(\sigma(T_1 \cup T_2))$, is as follows.

$$\begin{aligned}
&(\textbf{From the decomposition formula}) \\
&E(\sigma(T_1 \cup T_2)) = \sum_{i=1}^{n}\sum_{j=1}^{m} E(\sigma(T_1^i \cup T_2^j)) \\
&= \sum_{i=1}^{n}\sum_{j=1}^{m} (i \times j \times Pr(T_1^i \cup T_2^j | T) \times \sigma(T) \\
&(\textbf{By the conditional independence assumption}) \\
&= \sum_{i=1}^{n}\sum_{j=1}^{m} i \times j \times Pr(T_1^i | T) \times Pr(T_2^j | T) \times \sigma(T) \\
&= \sigma(T) \times \sum_{i=1}^{n} i \times Pr(T_1^i | T) \times (\sum_{j=1}^{m} j \times Pr(T_2^j | T)) \\
&= \sigma(T) \times \sum_{i=1}^{n} i \times Pr(T_1^i | T) \times (\sum_{j=1}^{m} j \times \frac{\sigma(T_2^j)}{\sigma(T)}) \\
&= \sigma(T) \times \sum_{i=1}^{n} i \times Pr(T_1^i | T) \times \frac{1}{\sigma(T)} \times (\sum_{j=1}^{m} j \times \sigma(T_2^j)) \\
&(\textbf{The decomposition of counts of } \mathbf{T_2},\ \sigma(\mathbf{T_2})) \\
&= \sigma(T) \times \sum_{i=1}^{n} i \times Pr(T_1^i | T) \times \frac{\sigma(T_2)}{\sigma(T)} \\
&(\textbf{The decomposition of counts of } \mathbf{T_1},\ \sigma(\mathbf{T_1})) \\
&= \sigma(T) \times \frac{\sigma(T_1)}{\sigma(T)} \times \frac{\sigma(T_2)}{\sigma(T)} \\
&= \sigma(T_1) \times \sigma(T_2)/\sigma(T)
\end{aligned}$$

□

In our approach, we will use the expected count of $T_1 \cup T_2$ as the estimator to estimate the count of $T_1 \cup T_2$, denoted as $\hat{\sigma}(T_1 \cup T_2) = \sigma(T_1) \times \sigma(T_2)/\sigma(T)$.

An important lemma that follows from this theorem is stated next.

Lemma 1 *Given two subtrees T_1 and T_2 which share a common subtree T, where*

$$|T| = min(|T_1|, |T_2|) - 1$$

then $\sigma(T_1 \cup T_2)$ can be estimated as follows:

$$\sigma(T_1 \cup T_2) = \frac{\sigma(T_1) \times \sigma(T_2)}{\sigma(T)}$$

[3]The coefficients in front of each term represents the number of choices one has to grow from T to T_1.

Proof Sketch: From the statement of the lemma we have $T_1 \cap T_2 = T$. Without loss of generality we can assume the following hold: $|T_1| = k$; $|T_2| = l$; $l < k$; and can therefore deduce $|T| = l - 1$.

Now, we pick a tree, $T_2^{'}$, such that: i) $T_2^{'}$ contains T_2; ii) $T_1 \cup T_2^{'} = T_1 \cup T_2$; and iii) $|T_1 \cap T_2^{'}|$, denoted as $|T^{'}| = k - 1$. We would like to note here that condition i) can be derived from the other two conditions but we explicitly state it here for the sake of expository simplicity. One can find such a tree by essentially starting with the intersection component (T) of T_1, and expanding it to all but one element of T_1 and then union it with T_2 to get $T_2^{'}$. Now from Theorem 1 and the way we pick $T_2^{'}$ we have:

$\sigma(T_1 \cup T_2) = \sigma(T_1 \cup T_2^{'}) = \frac{\sigma(T_1) \times \sigma(T_2^{'})}{\sigma(T^{'})} = \frac{\sigma(T_1) \times \sigma(T_2^{'})}{\sigma(T_1 \cap T_2^{'})}$ (1)

Now, we pick a tree, similar to the above instance, $T_2^{''}$, such that: i) $T_2^{''}$ contains T_2; ii) $T_2^{'} = T^{'} \cup T_2^{''}$; and iii) $|T^{'} \cap T_2^{''}|$, denoted as $|T^{''}| = k - 2$.

From Theorem 1 we know have: $\sigma(T_2^{'}) = \frac{\sigma(T_2^{''}) \times \sigma(T_l)}{\sigma(T^{'} \cap T_2^{''})}$ (2)

Substituting this (2) in (1) and canceling terms we have: $\sigma(T_1 \cup T_2) = \frac{\sigma(T_1) \times \sigma(T_2^{''})}{\sigma(T^{'} \cap T_2^{''})}$

Expanding $T^{'}$ we have: $\sigma(T_1 \cup T_2) = \frac{\sigma(T_1) \times \sigma(T_2^{''})}{\sigma(T_1 \cap T_2^{'} \cap T_2^{''})}$

Since we know that $T_2^{''}$ is a strict subset of $T_2^{'}$ we can reduce this to: $\sigma(T_1 \cup T_2) = \frac{\sigma(T_1) \times \sigma(T_2^{''})}{\sigma(T_1 \cap T_2^{''})}$

The rest of the proof relies on repeating this process till we get: $\sigma(T_1 \cup T_2) = \frac{\sigma(T_1) \times \sigma(T_2)}{\sigma(T_1 \cap T_2)}$

□

3.2 Recursive Decomposition Scheme

This decomposition is obtained directly from Lemma 1. Since each tree has at least two leaf nodes(if the root node has degree 1, it can also be considered a leaf node for our purposes), it is guaranteed that we can always obtain two subtrees(T_1 and T_2) of the original tree T by removing one or the other of its two leaf nodes respectively. If the size of T is k, then the size of T_1 and T_2 will be $(k - 1)$. Suppose the common part between T_1 and T_2 is T_3, then we can apply the above formula to estimate the selectivity of T, given the selectivity of T_1, T_2 and T_3.

This decomposition scheme ensures that the overlap between T_1 and T_2 is maximal and thus it ensures that correlation of occurrence is well captured. If T_1, T_2 are still too large that they are not in the lattice summary, then we execute the above decomposition process recursively, until we reach the brim of the lattice summary. We present an example of recursive decomposition in Figure 3a. The nodes in bold are chosen to be eliminated at each recursion. We use a 4-lattice in this example. Figure 4 presents the formal algorithm of the estimator.

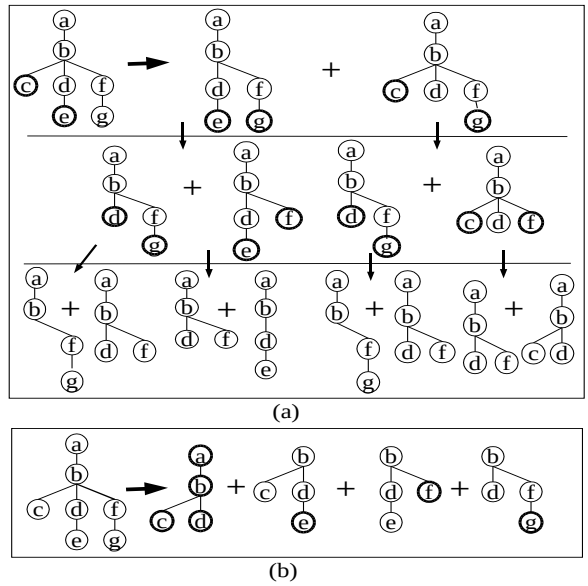


Figure 3. (a)Recursive Decomposition Scheme; (b)Fix-sized Decomposition Scheme

Voting Scheme Extension: We note that a tree may have more than two leaves. In this case the choice of leaf nodes for decomposition may result in different estimates. Correspondingly, we can have multiple estimations at each recursive step. In the optimization herein, we record all estimations at a given level and average them to obtain the resulting estimate to be

Algorithm: Estimate (T, L)
Input: T, an XML twig;
L, a k-lattice summary;
Output: $\hat{\sigma}$, selectivity estimation for T;
1. **if** $T.size \leq k$
look up T in L, and return the associated count;
2. **else**
pick a pair of T's nodes(v_1, v_2) having degree of 1;
remove v_1 from T to get T_1,
remove v_2 from T to get T_2;
evaluate $T_3 = T_1 \cap T_2$;
$\hat{\sigma} = \frac{Estimate(T_1,L) * Estimate(T_2,L)}{Estimate(T_3,L)}$

Figure 4. Algorithm for Recursive Decomposition Estimator

Input: T, an XML twig of size n and the lattice level k;
Output: D, a set of k-subtrees satisfying the condition in Theorem 2;
1. Order all nodes of T according to pre-order; i.e., $v_1, v_2, v_3, \ldots, v_n$
2. Choose the subtree t_1 consisting of the first k nodes from the node list; and label them as covered;
   ; t_1 must be a valid subtree
   Initialize T_c by t_1, add t_1 to D;
3. **for** each remained uncovered node v_i
4. pick a subtree t_i containing v_i as the rightmost node, and all other nodes are from T_c;
5. add v_i to T_c, label v_i as covered, and add t_i to D;
6. return D;

Figure 5. Fix-sized Decomposition Algorithm

used in the next step. Intuitively, we expect to avoid skewed estimates resulting from poor initial choices and expect that this optimization will relieve the error propagation during the course of the decomposition. Different voting schemes can be applied here accounting for higher order statistical moments and these are under evaluation. We will demonstrate the power of this optimization in Section 5.

3.3 Fix-sized Decomposition Scheme

Another approach to decomposition may be to use smaller, fix-sized subtrees(T_i's) to cover T. Toward this end we propose a step by step, progressively covering scheme using fix-sized subtrees.

Assume the maximal pattern in the summary is of size k, in other words, we keep the information of all subtrees no larger than k. We sort all nodes in the twig in pre-order fashion. Then we choose a k-subtree of T to cover the first k nodes. Let the covered portion of T be denoted as T_c. Then at each following step we cover a new node v using T_{new}, where all nodes except v of T_{new} are a subset of T_c. Correspondingly, we update T_c as the union of the previous T_c and T_{new}. As can be seen, T_c will grow progressively, until it covers all nodes in T. Also, it always holds that the common part between T_c and T_{new} is a $(k-1)$-subtree. Clearly, T can be covered by exactly $(size(T)-k+1)$ k-subtrees. The correlation between two subtree patterns is captured by their common part. In Figure 3(b) we present an example of this decomposition. Newly covered nodes are highlighted at each step. Figure 5 presents the formal algorithm of the fix-sized decomposition algorithm.

A more formal theorem and proof of correctness follows.

Lemma 2 *Given a rooted ordered labeled tree T of size n, it can be covered by $n-k+1$ of its subtrees of size k ($n > k$), i.e., $T_1, T_2, \ldots, T_i, \ldots, T_{n-k+1}$, such that $T_i \cap (\bigcup_{j=1}^{i-1} T_j)$ is a $(k-1)$-subtree.*

Proof: Let $T_i^c, 1 \leq i \leq n-k+1$, be the subtree of T covered by the first i subtrees, i.e. $T_i^c = \bigcup_{j=1}^{i} T_j$. We will prove that one can always find a subtree $T_{i+1}, i < n-k+1$ such that the size of the common subtree between T_i^c and T_{i+1} are $k-1$, and further, the size of T_i^c (the number of nodes) is $i+k-1$. We prove this statement by induction.

Base Step: For $i = 1$, the first k nodes in the prefix order of T always compose a k-subtree T_1. Clearly, the size of $T_1^c = T_1$ is k and one of the nodes in T_1 must have an edge to a node which is not covered by T_1. (Otherwise, the tree has been completely covered by T_1, i.e. $n = k$.)

We consider the following three cases to find a subtree T_2 which includes the new edge e_1.

Case 1: if the edge e_1 is adjacent to a non-leaf node of T_1, then we can build T_2 by simply removing one of the leaf node from T_1 and augment it with the new edge e_1;

Case 2: if the edge e_1 is adjacent to the only leaf node of T_1, (i.e., T_1 is a path), then we can build T_2 by removing the root node of T_1, extend it to include the new edge e_1;

Case 3: if the edge e_1 is adjacent to a leaf node of T_1 and T_1 has more than one leaf node, then we can build T_2 by removing one of leaf nodes not adjacent to e_1, and augment it with the new edge e_1.

Clearly, every possible, co-occurrence of e_1 and T_1 is covered in these three cases, and in each of them, we can find at least one k-subtree T_2 in T such that $T_2 \cap T_1^c = k-1$.

Induction Step: Next, assuming for $1 \leq i < n-k+1$, our statement holds. For $i+1$, we have a covered subtree T_i^c with $i+k-1$ nodes, we need to find a k-subtree T_{i+1} in T such that the size of the common subtree of T_{i+1} and T is $k-1$.

Clearly, if $i < n-k+1$, the size of the subtree T_i^c is $i+k-1 < n$. Therefore, one of the nodes in T_i^c must have an edge e_i adjacent to a node which is not covered by T_i^c. Also, each node in T_i^c has been covered by at least one k-subtree. Let the node in T_i^c adjacent to the edge e_i is a. Consider T_a be one of the k-subtrees, in $T_1, T_2, \cdots, T_i$, that contain the node a. We can then build T_{i+1} using the same method as in the base step (Note that we treat T_a as T_1 in this case).

As a result the size of the common subtree between T_{i+1} and T_i^c is $|T_{i+1} \cap T_i^c| = |T_{i+1} \cap T_a| = k-1$. The size of $T_{i+1}^c = T_i^c \cup T_{i+1} = T_i^c \cup \{e_i\}$ is $i+k-1+1 = (i+1)+k-1$.

For $i = n-k+1$, we have the size of T_{n-k+1}^c to be $n-k+1+k-1 = n = |T|$. This suggests we

covered the entire tree T, and therefore completes our proof. □

To obtain a selectivity estimate from this fix-sized decomposition covering scheme we rely on Lemma 3 described next.

Lemma 3 *Assume we have a twig query T decomposed into k-subtrees, i.e., $T_1, T_2, \ldots, T_i, \ldots, T_{n-k+1}$, and, $C_{i-1} = T_i \cap (\bigcup_{j=1}^{i-1} T_j), 2 < i \leq n-k+1$. Then the selectivity of T may be estimated as follows:*

$$\hat{\sigma}(T) = \frac{\prod_{i=1}^{n-k+1} \sigma(T_i)}{\prod_{j=1}^{n-k} \sigma(C_j)}$$

Proof: Let $T_i^c, 1 \leq i \leq n-k+1$, be the subtree of T covered by the first i subtrees, i.e. $T_i^c = \bigcup_{j=1}^{i} T_j$. Therefore, our final estimator will be as follows.

$$\hat{\sigma(T)} = \frac{\hat{\sigma}(T_{n-k}^c) \times \sigma(T_{n-k+1})}{\sigma(C_{n-k})}$$

$$= \frac{\hat{\sigma}(T_{n-k-1}^c) \times \sigma(T_{n-k}) \times \sigma(T_{n-k+1})}{\sigma(C_{n-k-1}) \times \sigma(C_{n-k})}$$

$$\cdots$$

$$= \frac{\hat{\sigma}(T_1) \times \sigma(T_2) \times \cdots \times \sigma(T_{n-k}) \times \sigma(T_{n-k+1})}{\sigma(C_1) \times \cdots \times \sigma(C_{n-k-1}) \times \sigma(C_{n-k})}$$

$$= \frac{\prod_{i=1}^{n-k+1} \sigma(T_i)}{\prod_{j=1}^{n-k} \sigma(C_j)}$$

□

Voting Scheme Extension: In this scheme again we have multiple choices when decomposing the twig query. If we keep all possible decompositions, then at the end, we can apply the same voting scheme described earlier. However, empirical study shows that the benefit of voting scheme to fix-sized decomposition scheme is not as significant as it is to the recursive decomposition estimator. Intuitively, the reason is that for the recursive decomposition estimator, we estimate the selectivity in a bottom-up fashion. The voting scheme can efficiently reduce the propagation of the error, especially in the first several recursions. On the other hand, for the fix-sized decomposition estimator, the voting is only applied at the very end so the errors have already propogated. As part of ongoing work we are evaluating if alternate voting schemes can help here.

3.4 Implications on Path Selectivity Estimation

The effectiveness of Markov model based XML path selectivity estimation approaches has been shown by [1]. Lemma 4 demonstrates that both of our decomposition approaches subsume the Markov model based XML path selectivity estimator as a special case.

Lemma 4 *Given a TreeLattice with an m-tree summary, the selectivity estimate for an XML path $T = t_1/t_2/t_3/\ldots/t_n, m < n$ is given by the Markov model based formula [9, 1]:*

$$\hat{\sigma}(T) = \sigma(t_{n-m+1} \ldots /t_{n-1}/t_n) \times \prod_{i=1}^{n-m} \frac{\sigma(t_{n-i-m+1}/\ldots/t_{n-i})}{\sigma(t_{n-i-m+2}/\ldots/t_{n-i})}$$

Proof: We prove this statement by induction. The following proof shows that for both the recursive decomposition estimator and fix-sized decomposition estimator the resulting estimate of an XML path expression will be the same as an estimate derived from the Markov model based approaches.

Base Step: For $n = m+1, T = t_1/t_2/t_3/\ldots/t_{m+1}$

According to our decomposition rules, T will be decomposed into two subpaths: $t_1/t_2/\ldots/t_m$, and $t_2/t_3/\ldots/t_{m+1}$; their common part is also a path, $t_2/t_3/\ldots/t_m$. Thus, we have the estimate for $\sigma(T)$ as follows:

$$\sigma(\hat{T}) = \frac{\sigma(t_1/t_2/\ldots/t_m) \times \sigma(t_2/t_3/\ldots/t_{m+1})}{\sigma(t_2/t_3/\ldots/t_m)}$$

Induction Step: Next, assuming for all $m < n \leq k$, the lemma holds. For $k+1$, we decompose T into two k-subpaths $p_1 = t_1/t_2/\ldots/t_k$ and $p_2 = t_2/t_3/\ldots/t_{k+1}$, and their common part is a $k-1$-subpath $p_3 = t_2/t_3/\ldots/t_k$, and according to the induction assumption, we may estimate the selectivity of p_1, p_2, p_3 as follows:

$$\hat{\sigma}(p_1) = \sigma(t_{k-m+1} \ldots /t_k) \times \prod_{i=1}^{k-m} \frac{\sigma(t_{k-i-m+1}/\ldots/t_{k-i})}{\sigma(t_{k-i-m+2}/\ldots/t_{k-i})}$$

$$\hat{\sigma}(p_2) = \sigma(t_{k-m+2} \ldots /t_{k+1}) \times \prod_{i=1}^{k-m} \frac{\sigma(t_{k-i-m+2}/\ldots/t_{k-i+1})}{\sigma(t_{k-i-m+3}/\ldots/t_{k-i+1})}$$

$$\hat{\sigma}(T) = \sigma(t_{k-m+1} \ldots /t_k) \times \prod_{i=1}^{k-m-1} \frac{\sigma(t_{k-i-m+1}/\ldots/t_{k-i})}{\sigma(t_{k-i-m+2}/\ldots/t_{k-i})}$$

Simplifying we get:

$$\hat{\sigma}(T) = \frac{\hat{\sigma}(p_1) \times \hat{\sigma}(p_2)}{\hat{\sigma}(p_3)}$$

$$= \sigma(t_{k-m+2} \ldots /t_{k+1}) \times \prod_{i=1}^{k-m+1} \frac{\sigma(t_{k-i-m+2}/\ldots/t_{k-i+1})}{\sigma(t_{k-i-m+3}/\ldots/t_{k-i+1})}$$

Therefore, the lemma holds. □

4 Lattice Summary and Implementation Details

In the previous section, we have shown how to use basic twigs to estimate the selectivity of large queries. These basic twigs form the statistics of the original XML data. Here, we will briefly discuss how we build and maintain such statistics in TreeLattice.

4.1 Building the Statistics

We rely on the lattice-based framework for enumerating all occurred subtree patterns as described in Freqt [2] and TreeMiner [18]. We build the statistics in a level-wise fashion. In other words, we collect the occurrence statistics of 1-subtrees first, then 2-subtrees, 3-subtrees, and so on. Since the number of all possible subtree patterns that occur in the XML data tree is potentially very large, we stop at a given level m, depending on our memory constraints. We use the resulting m-lattice as the statistics of the whole XML data tree.

4.2 Storing the Statistics

We have adopted the Freqt [2] tree mining algorithm to build the lattice summary of the XML data. The storing data structure should be very concise, and also, it should be very convenient when estimating the selectivity of the twig queries.

Aboulnaga *et al.* [1, 9] demonstrate that the use of a hash table to store the path statistics for estimating the selectivity of XML path queries is an efficient approach. We adopt a similar approach here to store the lattice statistics. We also considered a prefix-tree based approach, but found empirically, that chasing prefix trees are not very efficient for this purpose. The main problem with prefix trees was the fact that it required quite a bit of pointer chasing.

4.3 Pruning Derivable Statistics

The summary records the occurrence statistics of basic twigs. There exists redundancy in the summary that can be pruned to reduce the size of the summary. We first formally define the notion of a δ-derivable pattern in this context.

Definition 2 *A twig pattern is δ-derivable if and only if its true selectivity is within an error tolerance of δ. to its expected selectivity according to TreeLattice.*

By definition 2, 0-derivable patterns have the exact true selectivity as their expected selectivity. It is therefore safe to prune away the 0-derivable subtree patterns from the lattice summary without sacrificing the quality of estimations. As a result, we have more space to store more non-derivable patterns in the lattice.

Lemma 5 *The estimation given by TreeLattice with a lattice summary L is exactly the same as that when 0-derivable patterns are removed from L.*

The above idea can also be generalized to vary δ and thereby control the trade-off between accuracy and memory utilization. Figure 6 presents the formal algorithm of pruning δ-derivable patterns from a given lattice L_m.

Algorithm: Pruning δ-derivable patterns
Input: $L_m(m > 2)$ and δ
Output: L'_m, the compressed lattice summary
1. Initialize L'_m by 1 and 2-subtree patterns;
2. $k = 3$;
3. **While** $k \leq m$
4. **For** each k-subtree pattern p
5. Estimate $\sigma(p)$ and compute the estimation error e;
6. **if** $e > \delta$ **then** add p to L'_m;
7. $k++$;
8. return L'_m.

Figure 6. Pruning δ-derivable patterns algorithm

5 Experiments

In this section, we examine the performance of our proposed probabilistic decomposition approach for selectivity estimation on synthetic and real-life datasets. Our results verify the effectiveness of the proposed approaches, in terms of accuracy, response time and construction time of the summary for large XML datasets. We compare our approach with TreeSketches, a state-of-the-art scheme[14].

5.1 Experimental Setup

Datasets: We have used four publicly available datasets in our experiments: NASA, a real-life dataset converted from legacy flat-file format into XML and made available to the public; IMDB, a real-life dataset from the Internet Movie Database Project; PSD(Protein Sequence Database), a real-life dataset of integrated collection of functionally annotated protein sequences; and XMark, a synthetic dataset that models transactions in an on-line auction site. We would like to note that for the PSD dataset, since TreeSketches takes a long time to process, we present results on a sample. The main characteristics of the datasets are summarized in Table 1.

Query Workloads:

To generate the positive query (queries with non-zero selectivity) workloads we adopted the following strategy. We enumerate the set of all possible queries (subtrees) for a given dataset. Should the number of resulting subtrees per level be too large (evaluated at each level of the lattice) then we sample the patterns at a given level to formulate the workload. This level-by-level characterization and sampling also enable us

to evaluate the performance of our strategies, in particular their error propagation, in a controlled manner. The distribution of tree patterns at different levels is presented in Table 2. As can be seen from the table, for all four XML datasets, the number of patterns below level 4 is fairly low. This is because usually the label set of an XML document is small even the original document is huge. However, the number of tree patterns at higher levels will still blow up easily.

To generate the negative query workloads (queries with zero selectivity), we modify the positive workloads by randomly replacing node labels in the twig in accordance with their frequency of occurrence. Basically, more frequent labels are used for replacement more often so there is a greater chance for erroneous predictions. Then we filter those queries whose selectivity is above 0.

Experimental results show that TreeLattice almost always, greater than 99% of the time, returns the correct answer(0). This is as expected considering the error only happens when all subtrees of a twig Q are in the original data tree, but Q itself does not in the data tree, and it's easy to verify that this probability is very low. For the same workload TreeSketches reports a 100% accuracy since their algorithm is designed to do well on such queries.

Error Metric: We quantify the accuracy of estimations using the absolute error metric which is defined as $|\sigma - \hat{\sigma}|$ / max(s, σ), where the sanity bound s is used to avoid the artificially high percentages of low count queries. Following common practice [14, 15], we set s to be the 10-percentile of true query counts. In addition, if s is less than 10, then we set it as 10.

Hardware: All the experiments were conducted on a Pentium $1GHZ$ machine with $1GB$ RAM and running Linux 2.4.19.

5.2 Results

All results presented in this section assume the use of a 4-lattice as the summary in TreeLattice unless otherwise noted.

Accuracy of the Estimators: In this experiment, we evaluate the effectiveness of our proposed estimators in estimating the selectivity of complex twig queries with branching path expressions. Figures 7a-d, show the average selectivity estimation error on various workloads for Nasa, IMDB, PSD and XMark respectively. Note that in Figure 7(d) the Y-axis is in the log scale since the difference between the two approaches is extremely large. We see that for all workloads, the average estimation error of TreeSketches is extremely high. For all workloads except that at level 4, the errors are above 100%. For the workload at level 8, the error is even beyond 17000%. In contrast, TreeLattice performs fairly well on this dataset. For all workloads, the errors are below 25%.

For Nasa and XMark, all three estimators of TreeLattice outperform TreeSketches on all workloads tested. For PSD, all three estimators of TreeLattice outperform TreeSketches on workloads of relative small queries. When the query size exceeds 6, the fix-sized decomposition estimator is outperformed by TreeSketches, while the recursive decomposition estimators (with/without voting) are still able to outperform TreeSketches. For IMDB, TreeSketches is outperformed by the recursive decomposition estimator with voting on queries below level 7, and after that, TreeSketches is better.

The wide gap in performance especially on the XMark dataset was surprising to us and the author of TreeSketches[4] To examine this in more detail we plotted the cumulative probability distribution function of the errors in Figures 8a-d. The results are consistent with Figures 7 however they do reveal that for a small fraction of queries in our workload TreeSketches grossly overestimates the selectivity thus resulting in this wide disparity in performance on the XMark dataset. We discuss this behavior in the next section with a simple example.

After accounting for these outlier queries, the cumulative distribution function of errors still reveals that for the Nasa and XMark datasets, all three of our estimators clearly outperform TreeSketches. Our decomposition strategies and TreeSketches are comparable on the PSD dataset although the voting scheme performs slightly better. On IMDB, the fix-sized and recursive decomposition estimators are outperformed by TreeSketches while the recursive decomposition estimator with voting closely approaches the performance of TreeSketches.

Overall, the results show that TreeLattice is reasonably effective in summarizing the distribution of the underlying twigs. For all datasets, when the workload consists of relative small queries, TreeLattice estimates the selectivity quite accurately. When the query size keeps increasing, the quality of the estimates given by TreeLattice decreases correspondingly due to the error propagation. Overall, the recursive decomposition estimator augmented with voting scheme yields the best estimate in most cases, though the recursive decomposition estimator and fix-sized decomposition estimator can yield reasonable estimates also.

Response Time: In this experiment, we compare our estimators against TreeSketches in terms of the response time. Figures 9a-d, present the response time of different approaches on different workloads for Nasa, IMDB, PSD and XMark respectively. As can be seen from the figures, both the recursive decomposition estimator and fix-sized decomposition estimator execute

[4]We shared these results with them.

orders of magnitude faster than TreeSketches for relatively small queries. Furthermore, the fix-sized decomposition estimator is typically a couple of factors faster than recursive decomposition estimator, since the latter has the additional overhead of recursion. Both estimators scale very well as we increase the size of twig queries to be estimated.

When we apply the voting optimization on recursive decomposition estimator, the response time degrades. The degradation of response time becomes more significant as we increase the size of the twig queries. This is not surprising, since the number of possible decompositions combinatorially increases with the number of recursion levels. However, overall the voting-based recursive decomposition estimator is still able to outperform TreeSketches.

Summary Size and Construction Time: In this experiment, we evaluate the summary size and the cost of constructing the summary. Table 3 reports the summary sizes for the different datasets. We can see that the sizes of the summary are reasonably small compared to the original datasets for both approaches. For all TreeSketches experiments we set the summary size as 50KB. From the data, we see that we use much less space than 50KB for Nasa, PSD and XMark. For IMDB we use more than 50KB space.

Now let us look at the cost of constructing the summary. In TreeSketches, this is a very expensive operation since it involves a bottom-up clustering of similar substructures in the XML data tree. In contrast, in our approach, we just need to run the off-the-shelf efficient tree mining algorithms to collect the summary. Table 3 presents the summary construction time of two approaches for different datasets. The advantage of our approach over TreeSketches is striking–anywhere from one to two orders of magnitude improvement.

Effect of Pruning δ-derivable Patterns: Our pruning strategy of removing derivable patterns from the lattice summary, and enables us to store more non-derivable patterns in the summary. In this section we examine the effects of this approach. Figure 10(a) presents the space savings on different datasets when we prune 0-derivable patterns from their 4-lattice summary. The savings can be clearly seen for all four datasets. Specifically the savings on Nasa, PSD and XMark are very striking. This implies that the conditional independence assumption holds very well on these three datasets. In this case, the saved space from pruning 0-derivable patterns can be used to store the information of more non 0-derivable patterns.

For the Nasa dataset using the same space of storing all patterns in the 4-lattice(20KB), we can store all the non 0-derivable patterns of the 6-lattice. Figure 10(b) presents the experimental results when we use the new lattice summary for the Nasa dataset. We run recursive decomposition estimator with voting. Overall, we observe a significant improvement in estimation accuracy. Even for the workload consisting of relatively large queries(size=9), the error is below 15%. In contrast, the error of TreeSketches is well above 500% for the same workload. Considering TreeSketches uses much more space (50KB) than TreeLattice, the benefits are striking. The results for PSD and XMark are very similar and thus omitted here.

On the other hand, for IMDB, the space saving is not that significant compared with the other three datasets. This indirectly implies that the conditional independence assumption does not hold by and large for this dataset which may explain why our approach did not do as well as TreeSketches. If accuracy is tolerable one can resort to pruning more relaxed derivable patterns to reduce the size of the summary. This will sacrifice accuracy further, since there is some information loss. Figure 10(c) shows the space savings due to different δ-derivable patterns pruning. Figure 10(d) presents the estimation quality of using the corresponding summary structures at the different δ levels. When we increase δ, we will achieve more space savings. On the other hand, the estimation accuracy will degrade. However it seems the degradation is tolerable for δ 10% which incidentally result in a summary that is smaller than the TreeSketches summary (from Figure 10(c)).

Experimental Summary: Qualitatively, TreeLattice is very effective on estimating the selectivity of the twig queries. Recursive decomposition estimator and fix-sized decomposition estimator yield reasonable estimates in a very short time. The voting scheme can be applied if we want more accurate estimates. It is also more expensive to compute. Quantitatively, TreeLattice is orders of magnitude faster than TreeSketches both in terms of the time to estimate and in terms of the summary construction costs for the datasets we have evaluated to date. The pruning of δ-derivable patterns enables us to tradeoff accuracy when there is a memory budget.

5.3 Discussion

From the experimental results, we find that TreeLattice compared favorable with TreeSketches. Fundamentally we wanted to understand the reason behind this. We take an example that is drawn from our experimental results (suitably abstracted for expository simplicity).

Figure 11 illustrates the difference between the TreeSketches approach and our approach. A document T is presented in a concise format in Figure 11(a). The number associated with a node u represents the number of occurrences for the subtree which has u as its root. Therefore, the node c associated with a number, 3, in Figure 11(a) suggests the subtree where node c

Table 1. Dataset Characteristics

Dataset	Elements	File Size(MB)
Nasa	476646	23
IMDB	155898	7
XMark	565505	10
PSD	242014	4.5

has four e as its children appears three times.

Figure 11(b) is the synopsis generated by TreeSketches for the document T. The synopsis is a directed graph, $G(V, E)$. Each vertex in the vertex set V corresponds to a set of nodes with the same label in the original document. A vertex in V is labeled by the label shared by all nodes in the corresponding set. Each edge in the edge set E associates a weight. An edge (x, y) with the weight α represents that in average, each node in the original document in the set x has α children in the set y. For example, the vertex c in the Figure 11(b) corresponds to a set of four nodes in the document T: three of node c where each of them contains four node e as children, and one node c which has two node e. Therefore, the vertex corresponding the set in the synopsis has an edge to the vertex e, and the weight is $(3 \times 4 + 1 \times 2)/(3 + 1) = 3.5$.

Assume our twig query is Figure 11(d). Clearly, the true selectivity of this query in the document T is 48. Using the TreeSketches synopsis in Figure 11 to estimate the selectivity of this query, we will have $2 \times 1 \times 2 \times 3 \times 3.5 \times 2.3 = 96.6$. Compared with the true selectivity, the error produced by this method is more than 100%.

Consider applying our method to this document with 5-lattice. Basically, we record all the subtrees with their count (selectivity) up to 5-node trees. Figure 11 shows three subtrees recorded in the lattice. In our method, these trees can be used to estimate the selectivity of the twig query in Figure 11(d). Note that the tree T_1 is the common subtree of T_2 and T_3, and combine T_2 and T_3 will generate our interested twig. Therefore, our estimation will be $count(T_2) \times count(T_3)/count(T_1) = 57.6$. In this example, our method produces much more accurate results than the previous method.

Why does our approach perform much better in this example? Note that in the synopsis, an edge (x, y) with the weight α represents that in *average*, each node in the original document in the set x has α children in the set y. Assume we have n nodes in the set x, and the nodes have $\alpha_1, \cdots, \alpha_n$, children in the set y, respectively. In order to compress the XML document, the variance of α_i can be quite large. Further, to estimate the selectivity, the weights on multiple edges need to be multiplied. As we can see in our example, the error propagation in the multiplication can be very fast, and results in relatively large error.

Table 2. No. of Subtree Patterns

Level	Nasa	IMDB	PSD	XMark
1	61	88	64	27
2	82	120	78	40
3	213	877	289	147
4	688	9839	1313	503
5	2296	97780	6870	1333

Dataset	Time (Seconds)		Utilization (KiloBytes)	
	TreeLattice	TreeSketches	TreeLattice	TreeSketches
Nasa	59	7535	20	50
IMDB	53	942	212	50
PSD	39	614	33	50
XMark	540	79560	13	50

Table 3. Summary Construction Time and Memory Utilization

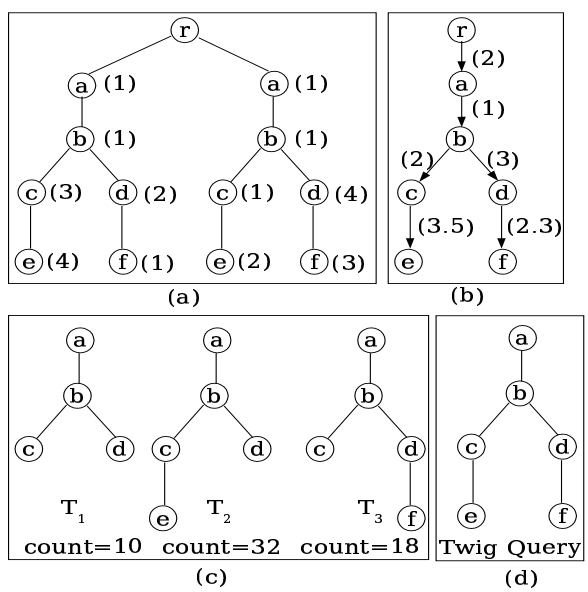
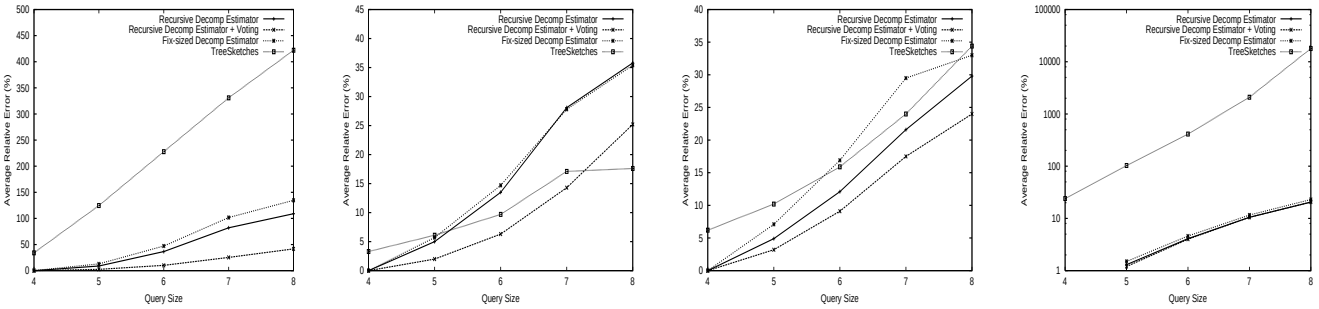
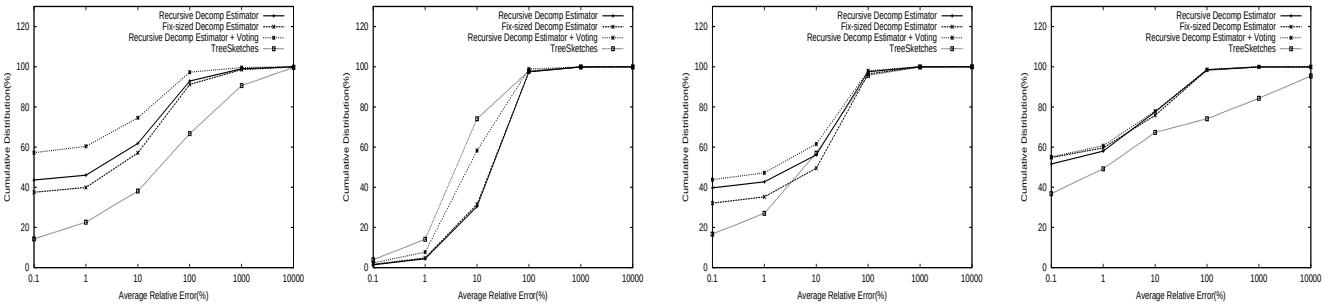


Figure 11. (a) Document T in Concise Format; (b) TreeSketches (c) Trees in Lattice (d) Twig Query



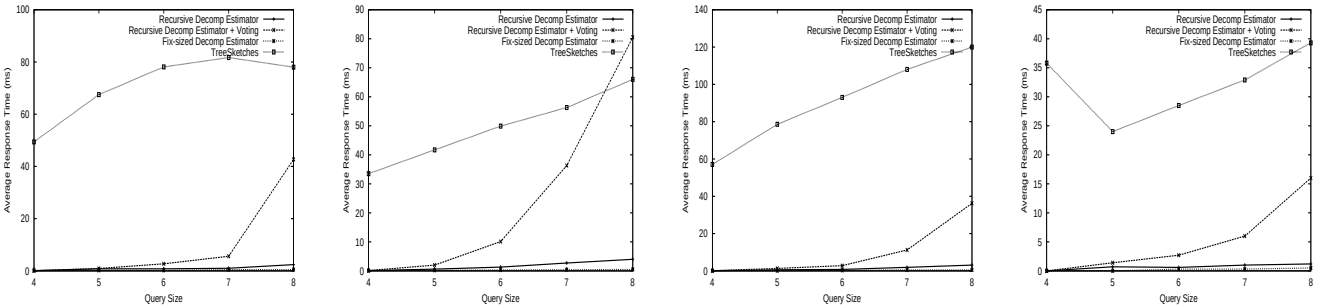
(a) (b) (c) (d)

Figure 7. Average selectivity estimation error:(a)Nasa (b)IMDB (c)PSD (d)XMark



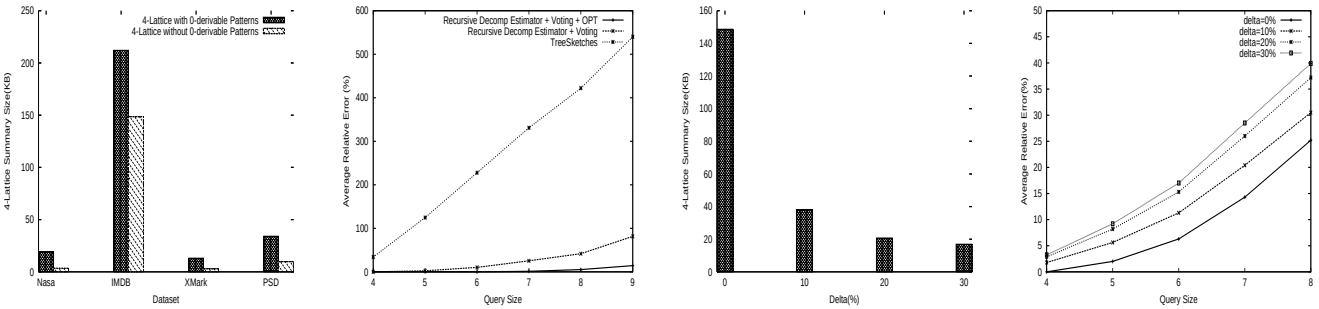
(a) (b) (c) (d)

Figure 8. Average Relative Error Distribution:(a)Nasa (b)IMDB (c)PSD (d)XMark



(a) (b) (c) (d)

Figure 9. Average response time:(a)Nasa (b)IMDB (c)PSD (d)XMark



(a) (b) (c) (d)

Figure 10. Derivable Patterns:(a)Removing 0-derivable Patterns (b)Average Relative Error(Nasa) (c)Summary Size(IMDB:Varying δ)(d)Estimation Quality(IMDB:Varying δ)

6 Conclusions and Future Work

In this paper, we have described a new approach, TreeLattice, to estimate the selectivity of twig queries. TreeLattice is shown to be comparable or better than TreeSketches in terms of estimation accuracy. Moreover, this new approach is also significantly faster both in terms of summary construction and in terms of selectivity estimation. Further, we have provided theoretical foundations for the estimation process and have also shown that the TreeLattice approach is a generalization of the successful Markov model based approach for XML path selectivity estimation.

In the future, we will study the following issues. First, we would like to extend the TreeLattice approach to work on the selectivity estimation for the twig queries with value predicates. Second, an error bound associated with the estimation would be very useful and we have made some initial progress towards this end. Third, we would also like to adapt TreeLattice, in a manner similar to XPathLearner where information learned from on-line workload can guide what is to be maintained in the summary structure.

References

[1] Ashraf Aboulnaga, Alaa R. Alameldeen, and Jeffrey F. Naughton. Estimating the selectivity of xml path expressions for internet scale applications. In *VLDB*, 2001.

[2] Tatsuya Asai and Kenji Abe *et al.* Efficient substructure discovery from large semi-structured data. In *SDM*, 2002.

[3] Zhiyuan Chen and H.V.Jagadish *et al.* Counting twig matches in a tree. In *ICDE*, 2001.

[4] Zhiyuan Chen, Flip Korn, and *et al.* Selectivity estimation for boolean queries. In *PODS*, 2000.

[5] Juliana Freire and Jayant R. Haritsa *et al.* Statix: Making xml count. In *SIGMOD*, 2002.

[6] H.V. Jagadish, Olga Kapitskaia, and *et al.* Multi-dimensional substring selectivity estimation. In *VLDB*, 1999.

[7] H.V. Jagadish, Raymond T.Ng, and *et al.* Substring selectivity estimation. In *PODS*, 1999.

[8] Wei Jiang, Haifeng Jiang, Hongjun Lu, and Jeffrey Xu Yu. Containment join size estimation: Models and methods. In *SIGMOD*, 2003.

[9] Lipyeow Lim and Min Wang *et al.* Xpathlearner: An on-line self-tuning markov histogram for xml path selectivity estimation. In *VLDB*, 2002.

[10] Jason McHugh and Jennifer Widom. Query optimization for xml. In *VLDB*, 1999.

[11] P.Krishnan, Jeffrey Scott Vitter, and Bala Iyer. Estimating alphanumeric selectivity in the presence of wildcards. In *SIGMOD*, 1996.

[12] Neoklis Polyzotis and Minos Garofalakis. Statistical synopses for graph-structured xml databases. In *SIGMOD*, 2002.

[13] Neoklis Polyzotis and Minos Garofalakis. Structure and value synopses for xml data graphs. In *VLDB*, 2002.

[14] Neoklis Polyzotis, Minos Garofalakis, and Yannis Ioannidis. Approximate xml query answers. In *SIGMOD*, 2004.

[15] Neoklis Polyzotis, Minos Garofalakis, and Yannis Ioannidis. Selectivity estimation for xml twigs. In *ICDE*, 2004.

[16] Wei Wang, Haifeng Jiang, Hongjun Lu, and Jeffrey Xu Yu. Bloom histogram: Path selectivity estimation for xml data with updates. In *VLDB*, 2004.

[17] Y. Wu, J. M. Patel, and H. Jagadish. Estimating answer sizes for xml queries. In *EDBT*, 2002.

[18] Mohammed Zaki. Efficiently mining frequent trees in a forest. In *SIGKDD*, 2002.