Counting Twig Matches in a Tree

Zhiyuan Chen
Cornell Univ
zhychen@cs.cornell.edu

H. V. Jagadish
Univ of Michigan
jag@umich.edu

Flip Korn
AT&T Labs–Research
flip@research.att.com

Nick Koudas
AT&T Labs–Research
koudas@research.att.com

S. Muthukrishnan
AT&T Labs–Research
muthu@research.att.com

Raymond Ng
Univ of British Columbia
rng@cs.ubc.ca

Divesh Srivastava
AT&T Labs–Research
divesh@research.att.com

Abstract

We describe efficient algorithms for accurately estimating the number of matches of a small node-labeled tree, i.e., a twig, in a large node-labeled tree, using a summary data structure. This problem is of interest for queries on XML and other hierarchical data, to provide query feedback and for cost-based query optimization. Our summary data structure scalably represents approximate frequency information about twiglets (i.e., small twigs) in the data tree. Given a twig query, the number of matches is estimated by creating a set of query twiglets, and combining two complementary approaches: Set Hashing, used to estimate the number of matches of each query twiglet, and Maximal Overlap, used to combine the query twiglet estimates into an estimate for the twig query. We propose several estimation algorithms that apply these approaches on query twiglets formed using variations on different twiglet decomposition techniques. We present an extensive experimental evaluation using several real XML data sets, with a variety of twig queries. Our results demonstrate that accurate and robust estimates can be achieved, even with limited space.

1 Introduction

Hierarchical organization of data is ubiquitous. Traditional examples include (UNIX and Windows) file systems, where files are organized hierarchically into directories and sub-directories; and yellow page business listings, which are categorized and sub-categorized hierarchically. More recent examples include XML data [1], where the element-subelement structure induces a natural hierarchy, and LDAP directories [8], where the set of directory entries are organized in a hierarchical name space. Such hierarchical data can be modeled as (large) node-labeled trees.

A natural way to query such hierarchically organized data is by using small node-labeled trees, referred to as *twigs*, that match portions of the hierarchical data. Such queries form an integral component of query languages proposed for XML (see, for example, [14, 7]), and for LDAP directories [10]. For example, the XML-QL query:[1]

```
    <book>
        <publisher>Morgan Kaufmann< /publisher>
        <year>1993< /year>
    < /book>
```

matches books published by Morgan Kaufmann in 1993, in the DBLP bibliography.[2] This query can be represented as a node-labeled tree, with the element tags “book”, “publisher”, and “year” as labels of non-leaf nodes in the tree, and the values “Morgan Kaufmann” and “1993” as labels of leaf nodes in the tree.

A fundamental problem in this context is to accurately and quickly estimate the number of matches of a twig query against the node-labeled data tree. Such estimation typically involves the use of a small summary data structure, instead of searching the entire database. This problem is relevant for providing users with quick feedback about their query, either before or along with returning query answers. Another use is in the cost-based optimization of such queries: knowing selectivities of various subqueries can help in identifying cheap query evaluation plans. In this paper, we address this important problem, and make the following contributions:

- We propose the use of a summary data structure, a *correlated subpath tree* (CST), that represents frequency information about twiglets (i.e., small twigs) in the data tree. This is achieved by: (i) maintaining, in the CST, frequently occurring subpaths in the data tree, and (ii) associating with each subpath in the CST, a small fixed-length signature of the subpath roots in the data tree. The signatures *scalably* (in accuracy as a function of database size) capture the correlations between the various subpaths in the data tree. As a result, the CST size is a small fraction of the database size.
- Given a twig query, the number of matches is estimated by first decomposing the query into a set of subquery pieces, referred to as *twiglets*, based on matches in the CST summary data structure, and then judiciously combining two complementary approaches: Set Hashing (SH), used to estimate the number of matches of each query twiglet in the data tree, and Maximal Overlap (MO), used to combine the query twiglet estimates into an estimate for the twig query. Based on these, we propose three promising estimation algorithms: maximal overlap with set hashing

[1] Really only the WHERE clause, since an XML-QL query can also restructure the matched data. We ignore restructuring since it is not relevant to this paper.

[2] http://www.informatik.uni-trier.de/ley/db/index.html.

(MOSH), piecewise-MOSH (PMOSH), and maximal set hashing (MSH).

- We present an extensive experimental evaluation of the proposed estimation algorithms using several real XML data sets, with a variety of twig queries. Our results demonstrate the scalable accuracy and robustness of the MOSH and MSH algorithms, even with limited space. On the DBLP data set, e.g., MSH has less than 20% relative error using only 1% of the space of the full data set; a greedy strategy, in contrast, tends to underestimate, and has close to 100% relative error.

To the best of our knowledge, ours is the first work on this timely topic. The rest of this paper is organized as follows. We start by discussing related prior work in Section 1.1. We formally define our problem in Section 2, and summarize our overall solution approach, including the CST summary data structure and the estimation techniques, in Section 3. We present details of the estimation strategies in Sections 4 and 5. Experimental results are presented in Section 6. Conclusions and directions for future work are outlined in Section 7.

1.1 Related Work

McHugh and Widom [13] describe Lore's cost-based query optimizer, which maintains statistics about subpaths of length $\leq k$, and uses it to infer selectivity estimates of longer *path queries*. However, since they do not maintain correlations between paths, their techniques do not allow them to accurately estimate the selectivity of *twig queries*, which are very natural in Lorel. Using our techniques, one could accurately estimate the selectivity of Lorel twig queries, potentially resulting in substantially better physical plans being generated.

In [12], the problem of substring selectivity estimation was introduced. An approach based on pruned suffix trees was presented wherein queries are parsed via a greedy strategy into substrings present in the pruned suffix tree, and the selectivities of these substrings are multiplied based on the independence assumption to derive selectivity estimates. In [11], the concept of conditioning based on maximal overlap parsing was introduced for improved estimation. Selectivity estimation of substrings over multiple attributes was considered in [15] and [9].

Set hashing has been used in a wide variety of applications, including estimating the size of transitive closure [5], finding Web page duplicates [2], and data mining [6]. Recently, the set hashing framework has also been considered by the present authors, who used it to capture co-occurrences of substrings across attributes and estimate the selectivity of boolean queries containing conjunctions, disjunctions and negations [4]. The present work applies to more complex structures than the one-dimensional strings considered in [4], and thus requires some ingenuity in how to decompose and prune the data tree into primitives that can be reconstructed. Furthermore, unlike substring queries, it is nontrivial how to parse (combine) a twig query into (from) twiglets, and thus we design clever techniques for doing so that obtain accurate estimates.

2 Problem Definition

Let Σ be an alphabet, let Σ^* be the set of strings of finite length on Σ, and $\Pi \subset \Sigma^*$ be a small set of strings. We are given a large rooted node-labeled tree $T = (V_T, E_T)$ where (i) non-leaf nodes are labeled with strings from Π, and (ii) leaf nodes are labeled with strings from Σ^*. An example of our input is an XML document; in this case, Π is the set of element tags and attribute names in the document. The query is a smaller, rooted, node-labeled tree $Q = (V_Q, E_Q)$, henceforth referred to as a *twig*. Non-leaf node labels of the query twig are from Π, and leaf node labels are from Σ^*. The goal is to determine the number of matches of Q in T.

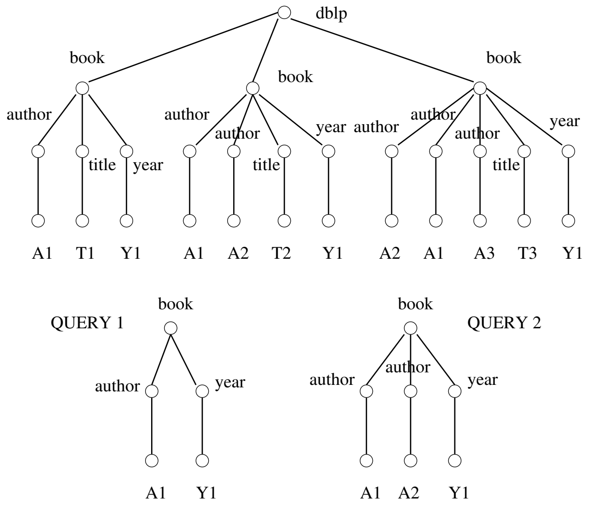


Figure 1: Example DBLP Data Tree and Twig Queries

Several versions of this problem are possible. For example, Figure 1 presents a node-labeled data tree, obtained by parsing an XML document that has three book elements. In the case of the left-most book element, sibling nodes have distinct labels, i.e., sibling labels form a set. In the other two book elements, some sibling nodes have identical labels, i.e., sibling labels form a multiset. Against this data tree, QUERY 1 (in Figure 1) has three matches. The number of matches of QUERY 2 depends on whether the twig query matching is performed in an unordered or an ordered fashion. In the unordered case, there are two matches, and in the ordered case, there is one match.

In the basic version of our problem, which we define here, and consider in the bulk of the paper, we will assume that sibling nodes in T and in Q have distinct labels; that is, we view sibling node labels as a set rather than a multiset, and that matching is unordered. In Section 5, we will consider its extension to the multiset, unordered case. Dealing with ordered matching is an interesting direction of future work.

The unordered matching is incorporated in our definition, below, of what constitutes a twig match.

Definition 1 (Twig Match) *A* match *of a twig query* $Q = (V_Q, E_Q)$ *in a node-labeled data tree* $T = (V_T, E_T)$ *is defined by a 1-1 mapping:*

$$f : V_Q \rightarrow V_T$$

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 problem of estimating the number of twig matches in a tree can now be stated as follows:

> Given a summary data structure T' corresponding to a node-labeled[3] data tree T, such that the size of T' is a small percentage of the size of T, and a twig query Q, estimate the total number of twig matches of Q in T, using only Q and the summary data structure T'.

3 Our Overall Approach

The problem of estimating the number of matches of a twig query in a data tree is a generalization of the problem of obtaining subpath count estimates when the data and query are both single-path trees. The analogy between subpaths and substrings leads us to look for a solution to our problem based on the approach known for estimating substring selectivity. The approach in [11, 12] is to keep statistics about frequently occurring substrings in a summary data structure, the pruned suffix tree (PST). A query string is parsed into pieces contained in the PST and the estimate for the entire query is synthesized from the counts of the parsed components.

The natural generalization of this approach to twig queries would keep frequently occurring twiglets (i.e., small twigs) in a summary data structure, and reconstruct the twig query from these building blocks. This presents two problems:

1. *The number of possible twiglets is very large*: For a string of length n, the number of possible substrings is quadratic in n, and these substrings can be stored in a suffix tree using space that is linear in n. However, for an n node tree, the number of possible twiglets is at least exponential in n because of the different shapes that can be generated, and it is not clear how these twiglets can be efficiently stored.
2. *The correlation among subpieces is rather complex*: Tree structured data is often highly correlated, *e.g.*, a node labeled "title" is likely to have a sibling node labeled "author". For a string of length n, only the correlation among successive pieces needs to be considered. For n node trees, the correlation among all possible subpaths rooted at the same node needs to be considered, and again this is at least exponential in the number of its children.

Hence, any method that *explicitly* stores statistics for a handful of twiglets has only a small sample of the statistical distribution of the occurrences of, and correlations between, the various subpaths.

In this paper, we adopt the simple, alternative approach of *implicitly* storing statistics for twiglets of the data tree by (i) maintaining count statistics about only a frequently occurring set of subpaths, and (ii) capturing the correlations among subpaths sharing the same root, by using a small fixed-length signature (the "set hash" signature [3, 5]) to represent the appearances of the root node on each subpath. The effect of our choice is to reduce the reliance on crude probabilistic formulae by conditioning on larger pieces. For example, suppose in the XML-QL query of Section 1 that the summary for the path "book.publisher.Morgan Kaufmann" stores the counts for the path and a signature representing the appearances of the node "book" on this path, and the same information is stored for path "book.year.1993". Then we can use the stored information to estimate the number of "book" nodes that appear on both paths, which is exactly the count for the twig query.

We refer to our summary data structure as a *correlated subpath tree* (CST). Several alternative approaches were considered before deciding to build signatures on the root nodes of subpaths. One such approach was to build signatures for all nodes on the subpath (including the root node), but this requires too much space to store signatures. (We would need to store three signatures instead of one for path "book.publisher.Morgan Kaufmann".)

3.1 The Summary Data Structure

The CST is constructed as follows. Let P be the set of all paths beginning at the root node in T, that is, the concatenations of each root-to-leaf node label sequence. We construct the path suffix tree from the paths in P, treating non-leaf node labels as *atomic*, while permitting substrings of leaf node labels to occur in the path suffix tree.[4] For example, given two root-to-leaf paths "dblp.book.author.Suciu" and "dblp.book.author.Sudarshan", the path suffix tree will contain nodes corresponding to each label ("dblp", "book", "author", "Suciu", "Sudarshan"), and to subpaths, such as "book.author", "author.Su", "uciu", "udarshan", etc. Note that paths such as "uthor.Suciu", and "author.uciu" do not occur in the path suffix tree.

Let $C^A(\sigma)$ be the path *appearance* count of the subpath σ, that is, the number of *paths* in T that contain σ as a subpath.[5] To reduce space usage, path suffix tree nodes v with small $C^A(\sigma)$ are thresholded, yielding the pruned subpath tree T'. After pruning, we replace the path appearance count value in each node v with the *presence* count $C^P(\sigma)$ of the subpath σ associated with v; this is the number of distinct *nodes* of T at which σ is rooted, and this count will be used in estimation. Thus, in our example above, T' may contain nodes corresponding to "dblp.book.author", "author.Su", "uciu" and "udarshan", but nodes corresponding to "dblp.book.author.Suciu" and "dblp.book.author.Sudarshan" may be pruned.

This pruned subpath tree, augmented with a small fixed-length signature (the "set hash" signature [3, 5]) in each node of T' corresponding to a non-leaf subpath, is our summary data structure, the correlated subpath tree (CST). Before describing the nature of this signature information, we motivate it by providing an overview of our estimation techniques.

3.2 An Overview of the Techniques

Let Q be a twig query for which we want to estimate the number of matches in a data tree T. Using the CST T', our estimation proceeds according to the following steps:

[3] There is no need to store signatures for leaf paths because there will be no sibling subpaths in the data and no correlations to capture for them.

[4] We do not explicitly store $Label(v)$ in each node of the path suffix tree; rather, we store a key used for accessing $Label(v)$ in a lookup table which is maintained in addition.

[5] We prune based on the number of *paths* containing σ. This will favor subpaths towards the root of T, since these node labels get repeated in many paths. This is necessary because, if we use the numbers of nodes rooting σ, then the root node of T will have a count of 1 and get pruned away.

Path Parsing: Independently parse each root-to-leaf path in Q into a set of (possibly overlapping) subpaths that have matches in T'. Let S denote the set of all such subpaths.

Twiglet Decomposition: For each twig node that is a branch node (*i.e.*, has two or more children), consider all subpaths in S rooted at the same node that pass through the branch node; each subtree induced by these subpaths is a query twiglet. Let S' denote the resulting set of (possibly overlapping) query twiglets.

Combination: Estimate the number of matches of Q using the augmented information in the nodes of T' by piecing together count estimates for the twiglets in S', using probabilistic estimation formulae.

Below we consider several options for each step, and describe the statistical information that needs to be maintained with the nodes of T' to enable the computations in the twiglet decomposition, and the combination steps. In Section 4, we give details and examples of the specific estimation methods.

3.3 Path Parsing Strategies

We consider two alternative strategies: *maximal* and *piecewise-maximal*. The maximal strategy parses each root-to-leaf path of Q into (overlapping) maximal subpaths, that is, the longest possible subpath associated with a node in T' until a mismatch occurs [11]. The piecewise-maximal strategy parses each segment in Q, where a "segment" is the sequence of nodes on the path between two consecutive root/branch/leaf nodes in Q. Each segment is independently parsed into (overlapping) maximal subpaths via the maximal strategy.

3.4 Motivating Set Hash Signatures

We assign a unique ID to every node v_T of the data tree T. Assume that for each node σ in the CST, we maintain the set of node IDs from T (denoted R_σ) that root the subpath σ. For example, given the subpath $\sigma = a.b.c$, we find the nodes v in T labeled a that have a child node labeled b, that in turn have a child node labeled c. For each such v, we record its node ID in a set which is maintained in the CST node associated with σ.

Note that the intersection of two subpath ID sets R_{σ_1} and R_{σ_2} yields the set of nodes from which the two subpaths emanate, that is, the nodes rooting the twiglet formed by subpaths σ_1 and σ_2. Given a twiglet composed of k subpaths $\sigma_1, \sigma_2, \ldots, \sigma_k$ emanating from the same node, the exact twiglet count can hence be determined by a k-way intersection: $|R_{\sigma_1} \cap \cdots \cap R_{\sigma_k}|$.

Unfortunately, having limited space prohibits explicit storage of, and performing k-way intersections of, the sets R_{σ_i}. To overcome this problem, we hash the sets R_{σ_i} and *estimate* intersections as needed, rather than to compute them exactly. For this, we rely on set hashing [3, 5] to create a small fixed-length signature $S_{R_{\sigma_i}}$ of each set R_{σ_i}.

3.5 Generating Set Hash Signatures

The set hash signature of a set R_{σ_i} is a vector of a small pre-determined size. We adopt the method from [4], summarized here.

Chen et al. [4] independently seed a linear hash function for each signature vector component, and generate the hash image $h(a)$ of each element $a \in R_{\sigma_i}$; the minimum $h(a)$ is recorded in the signature vector component. To reduce collisions, h must be designed to map elements into a range significantly larger than the domain. These set hash signatures $S_{R_{\sigma_i}}$ are stored with each node σ_i in the CST. We then use set hashing, as discussed below, to obtain an estimate of the k-way set intersection by "intersecting" the k signatures $S_{R_{\sigma_1}}, \ldots, S_{R_{\sigma_k}}$.

3.6 Estimating Twiglet Counts

Given a twiglet that is composed of k subpaths $\sigma_1, \sigma_2, \ldots, \sigma_k$ emanating from the same node, the set hash signatures of these k subpaths can be obtained directly from the CST. Estimating the count of twiglet matches can be done by manipulating signatures. We adopt the method from [4], and discuss these steps below.

Set resemblance estimation: The *set resemblance* (denoted ρ) of k sets is defined as the size of intersection of all these sets divided by the size of union of all sets. Given the k set hash signatures corresponding to the subpaths $\sigma_1, \sigma_2, \ldots, \sigma_k$ of the twiglet, the set resemblance ρ_k of the k sets $R_{\sigma_1}, \ldots, R_{\sigma_k}$ can be estimated by dividing the number of matching components in each of the signatures $S_{R_{\sigma_1}}, \ldots, S_{R_{\sigma_k}}$ by the length of each set hash signature.

Estimating intersection size via set hashing: The goal is to estimate $|R_{\sigma_1} \cap \cdots \cap R_{\sigma_k}|$.

Step 1: Compute ρ_k from the k signatures.

Step 2: We calculate $S_{R_{\sigma_1} \cup \cdots \cup R_{\sigma_k}}$, the signature of $R_{\sigma_1} \cup \cdots \cup R_{\sigma_k}$, from the signatures $S_{R_{\sigma_j}}$ for sets R_{σ_j} as follows. For any i, $1 \leq i \leq \ell$, (ℓ as the signature length), $S_{R_{\sigma_1} \cup \cdots \cup R_{\sigma_k}}[i] = \min\{S_{R_{\sigma_1}}[i], \cdots, S_{R_{\sigma_k}}[i]\}$. That is, for each component, we choose the minimal value of the signatures of all its sets.

Step 3: Say R_{σ_j} has the largest size among all R_{σ_i}'s.[6] Using $S_{R_{\sigma_j}}$ and $S_{R_{\sigma_1} \cup \cdots \cup R_{\sigma_k}}$ we estimate $\gamma = \frac{|R_{\sigma_j}|}{|R_{\sigma_1} \cup \cdots \cup R_{\sigma_k}|} = \frac{|R_{\sigma_j} \cap (R_{\sigma_1} \cup \cdots \cup R_{\sigma_k})|}{|R_{\sigma_j} \cup (R_{\sigma_1} \cup \cdots \cup R_{\sigma_k})|}$ as the resemblance of appropriate sets.

Step 4: We use the following formula: $|R_{\sigma_1} \cap \cdots \cap R_{\sigma_k}| = \rho_k \, |R_{\sigma_1} \cup \cdots \cup R_{\sigma_k}| = \frac{\rho_k |R_{\sigma_j}|}{\gamma}$. to now estimate $|R_{\sigma_1} \cap \cdots \cap R_{\sigma_k}|$, if $|R_{\sigma_j}|$ is kept explicitly, as indeed it is in our CST.

The details of how to construct the CST and the associated set hashes efficiently involve adopting existing techniques from previous papers [4] and are omitted.

3.7 Combining Counts

We start with a collection of (possibly overlapping) query twiglets for which we have obtained count estimates. To estimate the number of matches of the twig query, these query twiglet counts are "combined" in formulae based on the inclusion-exclusion principle, similar to the approach taken in [11]. The idea is to multiply the probabilities of the query twiglets and then divide by the counts of the overlapping portions so as not to count them more than once. An overlapping portion could be the null set

[6] Any set will do, but the largest gives the best accuracy.

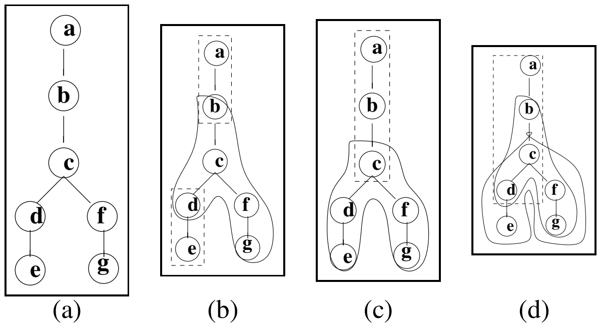


Figure 2: (a) example tree pattern (b) MOSH twiglets (c) PMOSH twiglets (d) MSH twiglets

(in which case we divide the count by 1), a single node, or a subtree. Note that the monotonicity property in the CST pruning guarantees that we can obtain a count for the subtwiglet (*i.e.*, overlapped portion). We refer to the combination process as MO conditioning.

4 Methods

We first describe "pure MO", which treats paths as twiglets, so correlations between paths are not captured. After that, there are three set hashing methods we consider. MOSH forms twiglets by combining paths that originate at the same node; correlations between paths within a twiglet are captured via set hashing. PMOSH and MSH are variants of MOSH that form twiglets differently to enhance the utilization of set hashing.

4.1 Pure MO

Our base-level strategy comes from a straightforward application of the MO parsing technique that was successfully employed in [11]. We henceforth refer to it as the "pure MO" technique. The specifics with respect to the general framework given in Section 3.2 are as follows:

Path Parsing: The paths are parsed according to the maximal strategy.

Twiglet Decomposition: The various subpaths obtained from maximal parsing are left as is, that is, $S = S'$. Thus, the twiglets that result are degenerate-case subpaths.

We shall demonstrate pure MO based on the twig example in Figure 2(a). The set of root-to-leaf paths of T is $\{a.b.c.d.e, a.b.c.f.g\}$. Suppose that MO parsing results in $\{a.b, b.c.d, d.e\}$ for path $a.b.c.d.e$ and in $\{a.b, b.c.f.g\}$ for path $a.b.c.f.g$. Thus, $S = \{a.b, b.c.d, d.e, b.c.f.g\}$. Let N denote the number of nodes in T and let $Pr(\sigma) = \frac{C^P(\sigma)}{N}$. The number of matches of Q in T, given by $C^P(\sigma)$, is estimated as follows:

$$N \times \frac{Pr(a.b) \times Pr(b.c.d) \times Pr(d.e) \times Pr(b.c.f.g)}{Pr(b) \times Pr(d) \times Pr(b.c)}$$

4.2 Maximal Overlap with Set Hashing (MOSH)

The goal of MOSH is to use set hashing to estimate the count of twiglet matches, so as to condition on fewer and larger (albeit approximate) overlapping counts. If all root-to-leaf paths in a twig query are present in the CST, the whole twig will form one twiglet, avoiding maximal overlap conditioning. The specifics with respect to the general framework given in Section 3.2 are as follows:

Path Parsing: The paths are parsed according to the maximal strategy.

Twiglet Decomposition: For each branch node in Q, and for each distinct starting point of the maximal subpath that goes through the branch node, use set hashing to obtain an estimate.

We illustrate using the example given in Figure 2(a). Again, suppose the independent MO parsing of the paths $a.b.c.d.e$ and $a.b.c.f.g$ results in $S = \{a.b, b.c.d, d.e, b.c.f.g\}$. Following the above steps, we identify the subpaths that go through the branch point c: $\{b.c.d, b.c.f.g\}$. Given the set hash signatures for these two subpaths, we can compute an estimate for the twiglet shown in Figure 2(b). We denote this estimate as $Pr(\text{sethash}(b.c.d, b.c.f.g))$. We thus have exact counts $Pr(a.b)$, $Pr(d.e)$, and an estimate $Pr(\text{sethash}(b.c.d, b.c.f.g))$. These can be combined using MO, to obtain $C^P(\sigma)$, as follows:

$$N \times \frac{Pr(a.b) \times Pr(\text{sethash}(b.c.d, b.c.f.g)) \times Pr(d.e)}{Pr(b) \times Pr(d)}$$

In the above example, the overlaps are subpaths, so the normalization probabilities are obtained directly from the CST. It is possible that overlaps themselves are subtrees, in which case the normalization probabilities also need to be estimated as well.

4.3 Piecewise MOSH (PMOSH)

It is possible that MOSH can reduce to pure MO when no two maximal subpaths in the parse of Q, that go through a branch node, start at the same node. Consider the following example. Given the example in Figure 2(a) and the parsing of the paths $a.b.c.d.e$ and $a.b.c.f.g$ into $S = \{a.b.c.d, c.d.e, a.b.c, b.c.f.g\}$, the maximal subpaths are thus $\{a.b.c.d, c.d.e, b.c.f.g\}$. For each branch point and each starting point, we list the maximal subpaths going through the branch point:

branch c, starting pt a = $\{a.b.c.d\}$
branch c, starting pt b = $\{b.c.f.g\}$
branch c, starting pt c = $\{c.d.e\}$

Unfortunately, we are unable to form a set hash intersection here, and must resort to pure MO to obtain the estimate.

We can make it less likely for this situation to occur by parsing each "segment" in Q, *i.e.*, the sequence of nodes on the path between two consecutive branch points, where the branch points are the start and end nodes of the path; alternatively, a segment may begin at the root, or may end at a leaf. Given the paths starting at each branch point, we can use set hashing to estimate the twiglet correlation. Finally, as in the pure MO strategy, we use MO to combine these estimates together. Using our example, we get that:

branch c, starting pt a = $\{a.b.c\}$
branch c, starting pt c = $\{c.d.e, c.f.g\}$

We combine via MO in the following formula (see Figure 2(c)) to estimate $C^P(\sigma)$:

$$N \times \frac{Pr(a.b.c) \times Pr(\text{sethash}(c.d.e, c.f.g))}{Pr(c)}$$

Note that the parsings are no longer maximal, since they may be cut short at the branch points. Due to the monotonicity property that every subsubpath of a subpath in the CST must also exist in the CST, shortening the paths is likely to find more paths intersecting at a branch point. The specifics with respect to the general framework given in Section 3.2 are as follows:

Path Parsing: The paths are parsed according to the piecewise-maximal strategy.

Twiglet Decomposition: Similar to MOSH.

4.4 Maximal Set Hashing (MSH)

The PMOSH strategy increases the number of subpaths that (i) intersect at the same branch point and (ii) begin at the same starting point, thus increasing the applicability of set hashing. However, it does so at the cost of shortening the parsed subpaths, which requires more MO conditioning and thus introduces more error. In general, MOSH forms deep but often skinny twiglets, while PMOSH forms bushy but often shallow twiglets. The goal of the maximal set hashing strategy (MSH) is to try to adapt the MOSH strategy to make more use of set hashing without shortening the path parsings as much. The specifics with respect to the general framework given in Section 3.2 are as follows:

Path Parsing: The paths are parsed according to the maximal strategy.

Twiglet Decomposition: The key idea, as in MOSH, is to collect subpath sets for each branch point and for each starting point of a maximal subpath that goes through the branch point. The difference is that, in any subpath set, one includes *all* subpaths that maximally begin at that starting point, not just the MO maximal subpaths; these are precisely the suffixes of the MO maximal subpaths that begin there.

Consider the example in Figure 2(a), again with paths $a.b.c.d.e$ and $a.b.c.f.g$ parsed into $S = \{a.b.c.d, c.d.e, a.b.c, b.c.f.g\}$, and thus the maximal subpaths are $\{a.b.c.d, c.d.e, b.c.f.g\}$. For each branch point and each starting point, we list the maximal subpaths going through the branch point:

branch c, starting pt a = $\{a.b.c.d\}$
branch c, starting pt b = $\{b.c.d, b.c.f.g\}$
branch c, starting pt c = $\{c.d.e, c.f.g\}$

These subpaths constitute S'. Note that we shorten subpath $a.b.c.d$ to $b.c.d$, and subpath $b.c.f.g$ to $c.f.g$ when we form the second and third twiglets in the example. But unlike PMOSH, the full subpaths $a.b.c.d$ and $b.c.f.g$ still participate in the formation of the first and second twiglets, respectively. Thus, the shortening of subpaths is not as much as PMOSH, and MSH often forms deep and bushy twiglets. Finally, we combine via MO in the following formula (see Figure 2(d)) to estimate $C^P(\sigma)$:

$$N \times \frac{Pr(a.b.c.d) \times Pr(\text{sethash}(b.c.d, b.c.f.g))}{Pr(b.c.d)} \times \frac{Pr(\text{sethash}(c.d.e, c.f.g))}{Pr(\text{sethash}(c.d, c.f.g))}$$

5 Dealing With Multisets

Up to this point, we have considered the basic version of the problem defined in Section 2, which assumes that no two sibling nodes in data tree T have duplicate labels, i.e., the children of any node in T can be described as a *set*. Here, we present techniques to handle the more general case when a node's children can be described as a *multiset*. In the multiset version of the problem, there are two different problem definitions that arise.

Definition 2 (Twig presence) *The number of* presences *of a twig Q in T is the cardinality of the* **set** *of nodes in T at which Q is rooted and there exists a 1-1 mapping $f : V_Q \to V_T$ such that if $f(u) = v$ for $u \in V_Q$ and $v \in V_T$, then $Label(u) = Label(v)$ and if $(u, u') \in E_Q$, then $(f(u), f(u')) \in E_T$, that is, the number of* distinct *root nodes of Q in T.* ∎

Definition 3 (Twig occurrence) *The number of* occurrences *of a twig Q in T is the cardinality of the* **multiset** *of nodes in T at which Q is rooted and there exists a 1-1 mapping $f : V_Q \to V_T$ such that if $f(u) = v$ for $u \in V_Q$ and $v \in V_T$, then $Label(u) = Label(v)$ and if $(u, u') \in E_Q$, then $(f(u), f(u')) \in E_T$, that is, the total number of possible mappings.* ∎

Note that the counts of twig presences and occurrences are the same in the set version of the problem. It is only when we deal with multisets that we must distinguish between these two.

We can estimate the counts for twig presences exactly the same way as we did for the basic set version of the problem. For twig occurrences, we preprocess the data to collect statistics as we did for the basic problem, but augment each node of CST T' with both the presence count $C^P(\sigma)$ and the occurrence count $C^O(\sigma)$ of the corresponding subpath σ. The basic idea for estimation of twig occurrences is to make the uniformity assumption on the *number of occurrences of a subpath σ for each presence of σ*, and estimate the occurrence count of a twiglet based on its presence count. This assumption is experimentally validated in Section 6. The procedure is as follows:

1. Path parsing and twiglet decomposition steps are exactly the same as for the basic problem.
2. For each twiglet t with k subpaths $\sigma_1, \cdots, \sigma_k$, we estimate the presence count $C^P(t)$ as we did for the basic problem. Then we estimate the occurrence count $C^O(t)$ as: $C^P(t) \times \prod_{1 \leq i \leq k} \frac{C^O(\sigma_i)}{C^P(\sigma_i)}$.
3. Combine estimate $C^O(t)$ for each twiglet t using the same probabilistic estimation formulae as we did in the basic problem.

For example, for the XML data tree shown in Figure 1, suppose the query is to count the total number of authors of books published in the year Y1, taking multiplicity into account. Suppose the query is parsed into one twiglet t as the query itself, and the presence count of t is estimated as 2.9. The presence counts for subpaths "book.author" and "book.year.Y1" are 3 each. The occurrence counts for these two subpaths are 6 and 3, respectively. Then we will estimate the occurrence count of t as $2.9 \times \frac{6}{3} \times \frac{3}{3} = 5.8$, which is very close to the occurrence count of 6 for the query.

6 Experimental Results

We implemented the four estimation algorithms: pure MO (referred to as MO in the figures for brevity), MOSH, PMOSH, and MSH of Section 4. In this section, we evaluate their relative performance experimentally. We also implemented two naive estimation algorithms for comparison: Greedy and Leaf. The Greedy algorithm differs from pure MO only in that Greedy parsing and combination [12] are used instead of MO. The Leaf algorithm ignores all path information and estimates the selectivity of each leaf string in the query individually using the MO parsing and combination techniques, then returns the product of estimates for each leaf as the twig query estimate; for instance, the count of the path query "book.author.Stonebraker" will be estimated as the MO estimate for "Stonebraker". Table 1 summarizes important properties of all the algorithms we implemented.

6.1 Experimental Setup

Data sets: We use two real XML data sets in our experiments. One is the DBLP bibliography. The data set size is 50MB, and consists of a tree with children rooted at `book` that itself has a variable number of children, e.g., `author`, `publisher`, etc.; see Figure 1. The other is the SWISS-PROT data set[7], which contains annotated protein sequences, including the sequences, annotations, authors, published places, citations, etc. The size is about 5 MB, but the structure is far more complex than the DBLP data set. Notice that both data sets contain duplicate sibling labels (e.g., a document may have multiple authors). This is the multiset counting problem.

Queries: Following [12, 11], we test both positive (meaning the twig does appear in the data) and negative (meaning the twig does not appear in the data) twig queries. Each workload consists of 1000 queries. Positive queries are randomly sampled from the data set. Each query consists of 2 to 5 paths, each having 2 to 4 internal nodes, and 1 to 4 characters from leaf node strings, with uniformly distributed probability. We also form a workload of *trivial* queries, each consisting of a single path.

The negative query workload consists of non-trivial queries having a true count of zero. The subpaths of each query are sampled from the data as for positive queries, and glued together.

Error metrics: For positive queries, we use *average relative error* and *average relative squared error* to measure the average accuracy. Let O_i be the true answer to a query and $O_i^{'}$ be our estimate. The average relative error for a workload of queries Q is defined as:

$$Error = \frac{1}{|Q|} \sum_{i \in Q} \frac{|O_i - O_i^{'}|}{O_i}$$

However, both relative error and absolute error have weaknesses in certain cases. For example, suppose query Q_1 has a real count of 10,000, and Q_2 has a real count of 100. Further suppose algorithm X's estimates for Q_1 and Q_2 are 5,000 and 50 respectively. Then Q_1 and Q_2 have the same relative error, but intuitively, the estimate for Q_1 is more erroneous. On the other hand, suppose algorithm Y's estimates for these two queries are 9,950 and 50. Again, intuitively the estimate for Q_2 should be more erroneous. However, both queries have the same absolute error. Therefore, we also use the average relative squared error, defined as:

$$Error = \frac{1}{|Q|} \sum_{i \in Q} \frac{(O_i - O_i^{'})^2}{O_i^{'}}$$

It is easy to check that the average relative squared error matches the intuition in both cases in the above example.

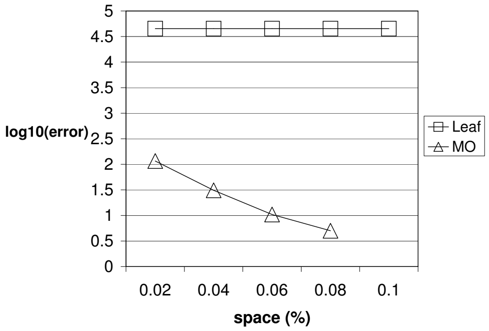


Figure 3: Average relative squared error in log10 scale for trivial queries, DBLP data set

For negative queries, following [11, 9], we use the root mean squared error to quantify the accuracy of the approach. This is defined as:

$$Error = \sqrt{\frac{1}{|Q|} \sum_{i \in Q} (O_i - O_i^{'})^2}$$

6.2 Accuracy for Trivial Queries

To examine whether path information is important for estimation, we compare the accuracy of the Leaf and the pure MO algorithm on trivial queries because they differ only in that MO stores path information whereas Leaf does not. Figure 3 shows the average relative squared error for the DBLP data set as we increase the space of the summary data structure. The increment is depicted as a percentage of the data set size. Results for the other data set and for other error metrics are similar and are omitted. The pure MO algorithm is up to a few orders of magnitude more accurate than the Leaf algorithm, which justifies the importance of using path information for estimation. This is also intuitive, for example, "Stonebraker" appears several times in "book.author.Stonebraker', but appears hundreds of times in "cite.Stonebraker".

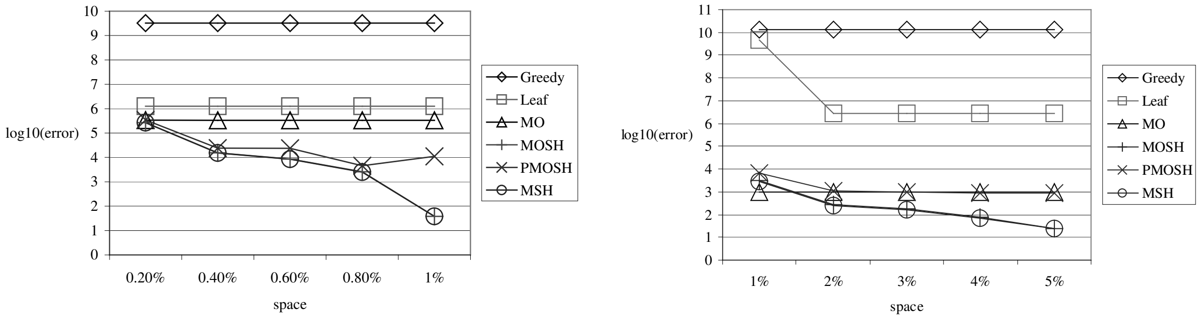
6.3 Accuracy for Positive Queries

In this section, we report the average error for positive, non-trivial queries as we vary the space of the summary data structure. We also report the distribution of queries with different *ratios* of the estimates to the real counts. We further examine the effect of twiglet decomposition technique by comparing the performance of MOSH, PMOSH, and MSH algorithms. To examine the scale-up quality of the algorithms, we also report how the error changes as we change the proportion of data extracted from the same data source.

[7] http://www.expasy.ch/sprot.

Name	Path Information	Correlation	Twiglets Formation	Combination Technique
Leaf	Not stored	Not stored	Single path	MO
Greedy	Stored	Not stored	Single path	Greedy
MO	Stored	Not stored	Single path	MO
MOSH	Stored	Stored	Deep but often skinny	MO
PMOSH	Stored	Stored	Bushy but often shallow	MO
MSH	Stored	Stored	Balance between deep and bushy	MO

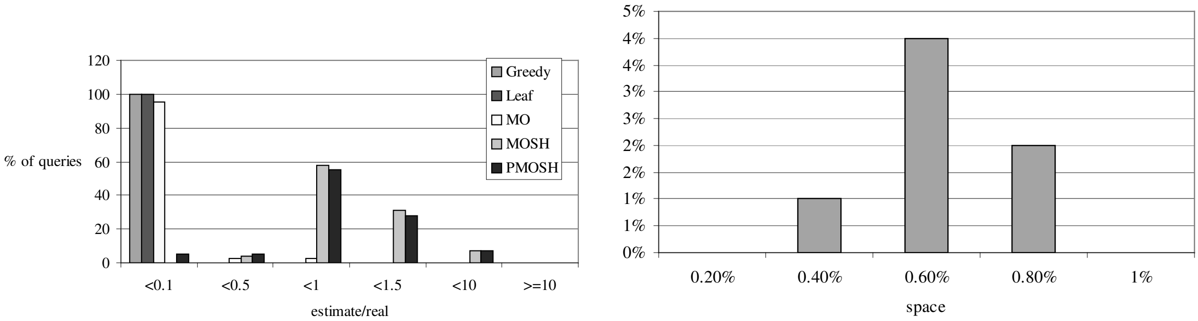
Table 1: Estimation Algorithms



(a) DBLP data set

(b) SWISS-PROT data set

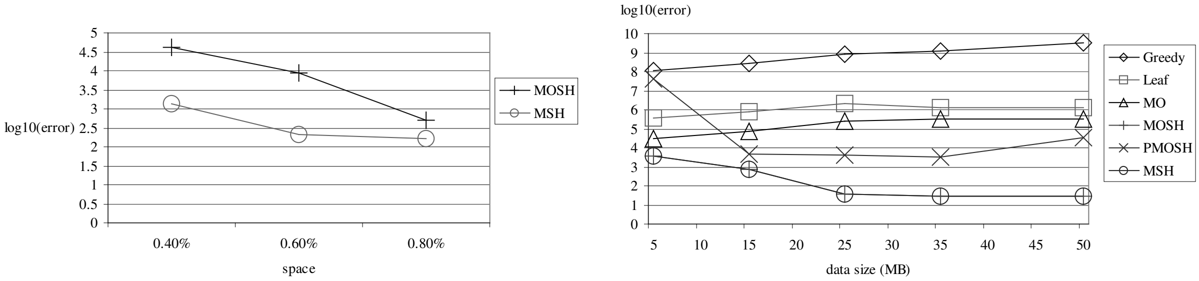
Figure 4: Average relative squared error in log10 scale of the algorithms as the space allowed for estimation increases for positive queries



(a) Distribution of ratio using 1% space

(b) Percentage of differently parsed queries

Figure 5: Differences between MOSH and MSH on the DBLP data set. The legend reads from left to right.



(a) For queries parsed differently by MOSH and MSH

(b) As the size of data extracted increases

Figure 6: Average relative squared error in log10 scale of the algorithms for the DBLP data set

Average error and ratio distribution for positive queries: Figure 4(a) shows the average relative squared error of all algorithms for the DBLP data set as the space increases. The results for the SWISS-PROT data set are shown in Figure 4(b). The average relative error for these data sets shows similar trends, and is omitted. The percentage of space used for the SWISS-PROT data set is larger because this data set is more complex than the DBLP data set and thus more space is needed for accurate estimation. For very small space, all three algorithms storing correlations perform as poorly as the other algorithms. This is not a surprise because so few nodes are kept in the summary data structure that all twiglets formed by these three algorithms are single path twiglets, i.e., these algorithms essentially degrade to pure MO.

As the space increases, however, the performance of MOSH and MSH improves substantially and soon outperform other algorithms. For example, for the DBLP data set, MOSH and MSH have 20% average relative error using 1% space; in contrast, Greedy, Leaf, and pure MO underestimate and have about 100% error. In terms of average relative squared error, MOSH and MSH outperform Greedy, Leaf, and pure MO by several orders of magnitude. More space translates to more nodes in the CSTs, and in turn to a higher probability for forming larger twiglets, i.e., allowing MOSH, PMOSH, and MSH to capture more correlations between subpaths in twig queries. The estimates of both Greedy and pure MO appear insensitive to the available space because both techniques soon have enough nodes in the CSTs to completely match every root-to-leaf query path as space increases, which is the best case for these algorithms. This justifies that storing correlations is crucial for accurate estimation. We also observe that pure MO is more accurate than Greedy, which shows that maximal overlap parsing and combination is better than the Greedy technique, as shown in [11, 9] for substring selectivity estimation. The performance of PMOSH is somewhat unstable, suggesting that the bushy but shallow twiglet decomposition technique is inappropriate.

Figure 5(a) shows the distribution of queries falling in specific ranges of the ratio of estimates to real counts. The results for the SWISS-PROT data set are similar and are omitted. The results for the MSH algorithm is not shown because it is very close to MOSH. The three algorithms not storing correlations (Greedy, pure MO, and Leaf) underestimate by more than an order of magnitude for more than 95% queries. In contrast, MOSH, PMOSH, and MSH estimate most queries within 50% of their real counts. MOSH and MSH also have no queries that are over- or under-estimated by an order of magnitude.

Twiglet decomposition techniques: The figures above show that PMOSH always underperforms MOSH and MSH. Therefore, forming bushy but often shallow twiglets does not work well in practice, because it sacrifices the depth of subpaths. Since MOSH and MSH often form the same twiglets, we further isolate those queries that are parsed into different twiglets by these two algorithms. Figure 5(b) shows the percentage of such queries on the DBLP data set. Results for the other data set are similar and are omitted. Figure 6(a) shows that for those differently parsed queries, MSH substantially outperforms MOSH, justifying that a balance between deeper and bushier twiglets is the best twiglet decomposition technique. However, the percentage of differently parsed queries is low (from 1% to 4%). Therefore, in practice, we could either choose MOSH to simplify implementation, or choose MSH for the best performance.

Scale-Up: We examine the scale-up property of all algorithms by extracting different sized data from the same data source. Figure 6(b) reports the average error as we extract 5 MB, 15 MB, 25 MB, 35 MB, and 50 MB data from the DBLP data set. The queries are positive and non-trivial, generated from each data set. For each data set, we use 2% space for the summary data structure. The results for the SWISS-PROT data set are similar and are omitted. The figure shows that MOSH and MSH have excellent scale-up property because their performance improves as the data size increases. The explanation is that the size of the unpruned data structure grows sublinearly with the data size, because more repetitions will occur in larger data sets. Since the space of the pruned summary data structure grows linearly with the data size, a larger part of the unpruned summary data structure will be stored for larger data sets. Thus, by storing correlations and judicially forming twiglets, MOSH and MSH take advantage of the larger summary data structure and perform better. In contrast, the performance of other algorithms do not have a clear trend of improving and even become worse in many cases, due to not storing correlations, or forming twiglets inappropriately.

6.4 Accuracy for Negative Queries

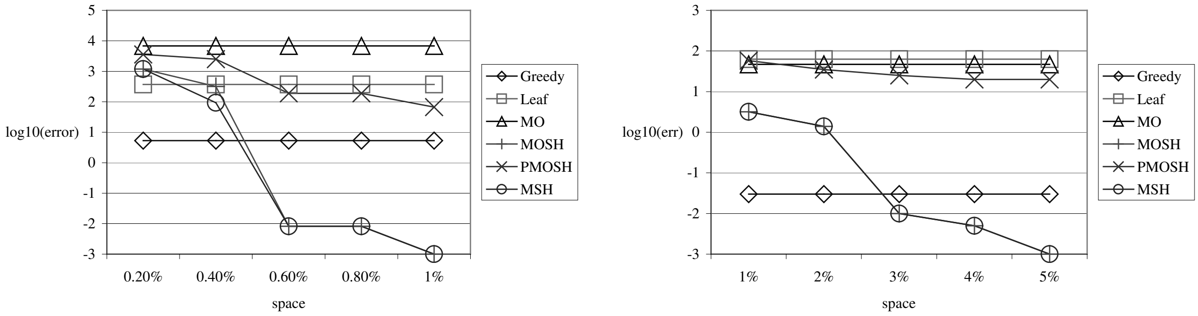
Figure 7 presents the results for negative queries. Greedy shows good performance and this is to be expected, since all the twiglets formed after parsing have very low counts and when multiplied to form an estimate, the resulting number is very close to the true value, i.e., zero. MOSH and MSH improve quickly as space increases and beat Greedy in the end. MO and Leaf appear much more inaccurate in this case, due to the amplification effect by conditioning on the overlapping paths, which have very small counts. PMOSH also has large error, due to the inappropriate twiglets forming technique.

6.5 Time

All our experiments were performed on a Pentium II, 350MHz, with 128MB of memory. It takes less than 10 minutes to construct and prune the CSTs for all algorithms and data sets. Estimations take about a millisecond for each algorithm. In all, construction and estimation are very fast.

6.6 Summary

We find that four issues are important for accurate estimation for twig queries: path information, correlations, twiglet decomposition, and parsing and combination technique. MOSH and MSH address all these issues and provide accurate and robust estimates for various types of data sets, various query types, and have excellent scale-up property, with very limited space. Leaf, Greedy, pure MO, and PMOSH each fail to address some of the above issues and do not provide accurate and robust estimates. The choice between MOSH and MSH depends on whether we want to simplify implementation or want the best performance.



(a) DBLP data set

(b) SWISS-PROT data set

Figure 7: Average root mean square error in log10 scale of the algorithms for negative queries as the space allowed for estimation increases.

7 Conclusions and Future Work

We described efficient algorithms for accurately estimating the number of matches of a twig query in a large node-labeled data tree, using the CST summary data structure, for unordered sets and multisets of sibling node labels in the data tree. In particular, we demonstrated the accuracy, robustness, and scalability of the MOSH and MSH estimation algorithms for various types of data sets and query types, even with very limited space.

There are several interesting directions of research that we are currently exploring. First, how do our techniques deal with matching twigs in an *ordered* node-labeled data tree? A possible solution is to enhance the set hashing approach such that, with each set hash value, we also keep track of the associated node ID. By assigning node IDs to the nodes in the data tree in a depth-first traversal order, and additionally checking that node IDs associated with paths emanating from a given branch node are in the desired order, we should be able to obtain good estimates for the ordered twig matching problem. Combining order with multisets is also important.

Another important direction of research is to understand how the techniques presented in this paper can be extended to estimate twigs with wildcards. A possible solution here involves the use of a special symbol "*" in the CST that matches arbitrarily long subpaths in the data tree, and using maximal parsing strategies on the twig query paths to include the "*"s appearing in the query. Doing so in a space-efficient manner, while ensuring accurate estimates, is a challenging problem.

Acknowledgements

H. V. Jagadish was supported in part by NSF under grant IIS-0002356.

References

[1] T. Bray, J. Paoli, and C. M. Sperberg-McQueen. Extensible markup language (XML) 1.0. W3C Recommendation. Available at http://www.w3.org/TR/1998/REC-xml-19980210, Feb. 1998.

[2] A. Broder. On the Resemblance and Containment of Documents. *IEEE SEQUENCES '97*, pages 21–29, 1998.

[3] A. Broder, M. Charikar, A. Frieze, and M. Mitzenmacher. Minwise Independent Permutations. *Proceedings of STOC*, pages 327–336, 1998.

[4] Z. Chen, F. Korn, N. Koudas, and S. Muthukrishnan. Selectivity estimation for boolean queries. In *Proceedings of the ACM Symposium on Principles of Database Systems*, 2000.

[5] E. Cohen. Size-Estimation Framework With Applications To Transitive Closure And Reachability. *Journal Of Comput. Syst. Sciences, 55*, pages 441–453, 1997.

[6] E. Cohen, M. Datar, S. Fujiwara, A. Gionis, P. Indyk, R. . Motwani, J. Ullman, and C. Yang. Finding Interesting Associations Without Support Pruning. *Proceedings of the 16th Annual IEEE Conference on Data Engineering (ICDE 2000)*, page to appear, Feb. 2000.

[7] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. XML-QL: A query language for XML. Submission to the World Wide Web Consortium 19-August-1998. Available from http://www.w3.org/TR/NOTE-xml-ql., 1998.

[8] T. Howes, M. Smith, and G. S. Good. *Understanding and deploying LDAP Directory Services*. Macmillan Technical Publishing, Indianapolis, Indiana, 1999.

[9] H. V. Jagadish, O. Kapitskaia, R. T. Ng, and D. Srivastava. Multi-dimensional substring selectivity estimation. In *Proceedings of the International Conference on Very Large Databases*, Edinburgh, Scotland, UK, Sept. 1999.

[10] H. V. Jagadish, L. V. S. Lakshmanan, T. Milo, D. Srivastava, and D. Vista. Querying network directories. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, Philadelphia, PA, June 1999.

[11] H. V. Jagadish, R. T. Ng, and D. Srivastava. Substring selectivity estimation. In *Proceedings of the ACM Symposium on Principles of Database Systems*, Philadelphia, PA, June 1999.

[12] P. Krishnan, J. S. Vitter, and B. Iyer. Estimating alphanumeric selectivity in the presence of wildcards. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 282–293, 1996.

[13] J. McHugh and J. Widom. Query optimization for XML. In *Proceedings of the International Conference on Very Large Databases*, pages 315–326, 1999.

[14] J. Robie, J. Lapp, and D. Schach. XML query language (XQL). Available from http://www.w3.org/TandS/QL/QL98/pp/xql.html.

[15] M. Wang, J. S. Vitter, and B. Iyer. Selectivity estimation in the presence of alphanumeric correlations. In *Proceedings of the IEEE International Conference on Data Engineering*, pages 169–180, 1997.