LCS-TRIM: Dynamic Programming Meets XML Indexing and Querying

Shirish Tatikonda, Srinivasan Parthasarathy, and Matthew Goyder
Department of Computer Science and Engineering
The Ohio State University, Columbus, OH 43210, USA

ABSTRACT

In this article, we propose a new approach for querying and indexing a database of trees with specific applications to XML datasets. Our approach relies on representing both the queries and the data using a sequential encoding and then subsequently employing an innovative variant of the longest common subsequence (LCS) matching algorithm to retrieve the desired results. A key innovation here is the use of a series of inter-linked early pruning steps, coupled with a simple index structure that enable us to reduce the search space and eliminate a large number of false positive matches prior to applying the more expensive LCS matching algorithm. Additionally, we also present mechanisms that enable the user to specify constraints on the retrieved output and show how such constraints can be pushed deep into the retrieval process, leading to improved response times. Mechanisms supporting the retrieval of approximate matches are also supported. When compared with state-of-the-art approaches, the query processing time of our algorithms is shown to be up to two to three orders of magnitude faster on several real datasets on realistic query workloads. Finally, we show that our approach is suitable for emerging multi-core server architectures when retrieving data for more expensive queries.

1. INTRODUCTION

Recently, the use of structured and semi-structured data has been increasing at a tremendous pace. A wide range of applications ranging from bioinformatics to social networks, from the World Wide Web to computational linguistics, are now generating and processing a large amount of semi-structured data. Languages such as XML have become a de-facto standard for semi-structured information exchange in many commercial and scientific applications. With standards like MPEG-7, a variety of multimedia features are now represented in a semi-structured form allowing for a uniform representation and content based image retrieval. With this increasing use of semi-structured data the need for maintaining and querying such data efficiently is also growing. Much research has focused on the efficient retrieval of semi-structured data from large databases. Several query languages, such as XPath [9], XQuery [5], and Quilt [6], have been developed with a view to specify complex structured queries on structured data.

In the context of XML, the semi-structured nature of the data and the queries themselves can often be modeled as trees. As a result the problem of querying such semi-structured datasets can be re-formulated as the one where the objective is to match a query tree structure as an embedded subtree within a database of trees. One approach researchers have taken to address this problem is to break down structured twigs into simple paths on which the database is indexed[2, 7, 8, 12, 21]. A potentially expensive join operation is then performed on the results from the path-based queries. More recent research has focused on *holistic* processing of the twig query[14, 26, 29]. Approaches such as PRIX[26] and $ViST$[29] rely on a representation that converts both the queries and the data into a sequence based representation. These approaches rely on subsequence matching of the query sequence with the database tree sequences to determine the candidate matches and then refine the candidates until the exact matches are identified. The key here is to minimize the number of false positive candidates and to efficiently compute the matches, which is the underlying principle motivating our work.

In this work, we explore new sequence-based methods for indexing and querying a database of tree structures. We propose two new tree encoding methods that are based on Prüfer Sequences and Depth First-order Sequences. We then *modify* the classic longest common subsequence (LCS)-based dynamic programming approach to find the location of *all* subsequence matches and use it for our purpose. We develop a novel structure matching algorithm, to prune false positive subsequence matches, that scans the twig *only once*. We design and evaluate two optimizations to reduce the overhead thereby making our algorithms efficient. We then present a unified algorithm that intelligently performs both the subsequence and structure matching simultaneously resulting in the early pruning of potential false positive matches. Additionally we also present mechanisms that enable the user to specify constraints on the retrieved output and show how such constraints can be pushed deep into the retrieval process leading to improved response times. Mechanisms supporting the retrieval of approximate matches are also supported.

Unlike the state-of-the-art approaches, our LCS based TRee Indexing and Matching (LCS-$TRIM$) algorithm op-

VLDB '07, September 23-28, 2007, Vienna, Austria.

erates on simple matrices that are built on-the-fly. Such an approach results in small memory footprints, good locality, and is amenable to parallelization on emerging multi-core architectures. The optimizations we adopt and the way we leverage our sequencing methods to efficiently find the subsequences distinguish our approach from the classic simple LCS problem. Our empirical analysis showed that our algorithms are very efficient compared to the state-of-the-art approach PRIX. In fact, we achieve up to three orders of magnitude speedup compared to state-of-the-art holistic approaches on a range of datasets and query workloads.

To reiterate the key contributions of this work are: **First**, we propose two new sequence representations for labeled rooted trees that are more concise and space-efficient when compared with other sequencing methods. **Second**, we develop a new dynamic programming based approach for finding *all* occurrences of a subsequence within a single sequence and by extension within a database of sequences. **Third**, we develop several novel optimizations, that eliminate false positive candidates early in the search process of likely matches making our approach extremely efficient in finding the desired set of twig matches. **Fourth**, we present several extensions to the above, allowing the user to specify constraints that can be pushed deep into the process thereby making the search even more efficient, parallelizing the search thereby facilitating the implementation of the approach on emerging CMP architectures and finally allowing for approximate results to be returned when specified by the user to do so.

2. BACKGROUND

Most of the till date research on tree indexing can broadly be divided into two main classes – *path-based approaches* and *holistic approaches*. The path-based approaches first decompose the given query into sub queries, which are usually individual root-to-leaf paths [2, 7, 8, 12, 21]. The solutions for these sub queries are then merged to form the final answer. A *range encoding* mechanism is usually employed to encode each node by its positional representation within the XML document such as (*start, end, depth*) [23].

Holistic approaches are typically shown to outperform the path-based approaches as they avoid the expensive join operation by treating the whole twig query as the base unit for query processing. A subset from this class of algorithms are based on TwigStack [4, 7, 16, 17, 20]. They maintain a stack against each node in the twig query that stores the partial results of some sub queries. They are often referred to as *holistic twig joins* as they perform global query matching. A variety of indexes such as B^+Tree [7], XB-Tree [4], XR-Tree [16, 17] are used to speedup the query processing. Typical probes to the index are in the form of *findAncestors* and *findDescendants*, which are answered efficiently by skipping some of the unnecessary data. A comparative study on the effectiveness of these indexes is done by Li *et al* [20].

Another subset of holistic algorithms recast the tree matching problem in to a subsequence matching problem by encoding the trees as sequences and by operating on sequences thereafter. In order to transform trees into sequences, these algorithms rely on well-known tree traversals such as pre-order and post-order. Since only the tree structure is captured by these traversals, they are supplemented with node label information so that they uniquely represent the trees. Resulting sequences with both structure and label information are referred to as *sequentures*. The query processing starts by finding the subsequence matches for the twig's sequence – the most important step in this class of algorithms [23]. One or more post-processing steps are then employed to filter the subsequence matches which do not correspond to actual twig matches.

Wang *et al* proposed $ViST$ [29] that relies on sequentures derived from pre-order traversals. PRIX [26] proposed by Rao *et al* uses Prüfer sequences which are constructed from post-order traversals. Wang and Meng proposed various performance-oriented principles to guide the tree sequencing process [30]. A limitation here is that for dynamic datasets the sequencing process cannot be determined apriori and for large datasets computing the optimal (constraint) sequencing may be expensive. In order to find the subsequence matches, these algorithms make use of B^+Tree based indexes. The tree nodes are indexed using positional representation such as *range encoding* [26, 29]. A series of *range queries* are then issued to the index to find the matches where each range query obtains the set of all descendants of a given node. Moro *et al* showed that PRIX and $ViST$ correspond to fixed plans of an Index Nested Loop Join (INLJ) [23]. They then compared all plans of INLJ with TwigStack and showed that TwigStack typically performed better and more robustly. Our algorithm LCS-TRIM is also a sequence-based holistic approach. Each tree in the data set is represented using two sequences which capture mutually exclusive but complementing information, tree structure and node labels. This leads to a space efficient representation when compared to competing sequencing methods (see Section 3.1). In contrast to $ViST$ and $PRIX$, we employ a dynamic programming based approach that does not use any index, in order to find the subsequences. We leverage the index in reducing the search space in early stages of query processing. Furthermore in order to filter out a false positive subsequence, PRIX employs a sequence of post-processing steps, each of which makes a complete pass on the query sequence whereas LCS-TRIM makes *a single pass* to filter out the false positive subsequence.

Zezula *et al* proposed methods which are based on tree signatures, which are constructed from both pre- and post-order traversals [35, 36]. While they also determine the twig matches by employing a dynamic programming based approach, LCS-TRIM differs from these methods in many different ways. *First*, our sequences are much more compact than their extended signatures because of *firstFollowing* and *firstAncestor* nodes. *Second*, the way LCS-TRIM identifies the false positive subsequences is a major difference from their method. *Third*, our simple inverted index complements our tree matching algorithm in early-pruning of the search space. *Fourth*, our novel optimizations in LCS-TRIM greatly reduce the computation overhead. *Finally*, embedding of pruning steps into the matching process is a key innovation we have that is not in the methods proposed by Zezula *et al*. These differences directly translate into tremendous performance improvement in query processing (see Section 4).

There exists few other indexing algorithms which are based on structural indexes and navigational mechanisms. They directly or indirectly rely on TwigStack algorithms. The navigational methods navigates through the database trees sequentially by using finite state machines, which store the partial results [1, 23, 34]. Structural indexes provide concise summaries for path structures and frequent query patterns

T	Database tree with n nodes
Q	Twig query with m nodes
NPS_T	Numbered Prüfer sequence of T
LS_T	Label sequence of T
$NPS_T[k]$	k^{th} entry in NPS_T (similarly $LS_T[k]$)
$CPS(T)$	(NPS_T, LS_T)
$R[i,j]$	An entry in the LCS matrix R
SM	A subsequence match (i_1,...,i_m)

Table 1: List of notation

with branches and wildcards ('*', '//') [8, 10, 12, 19]. However, a twig query that is not in the indexed paths and query patterns has to again rely on the join operation.

3. APPROACH

Our approach for indexing and querying the tree databases has three main parts: *data representation*, *tree matching*, and *tree indexing*. First, the database trees and the given queries are represented as sequences. We show two different tree sequencing methods (see Section 3.1). The tree matching algorithm then finds the set of all matches for a given query in the database (see Section 3.2). The entire process of querying is made efficient by employing an index (see Section 3.3) and few optimizations to the tree matching algorithm (see Section 3.4).

3.1 Data Representation

We now describe two tree sequencing methods, which provide a bijection between rooted, ordered, labeled trees and sequences. Both *prüfer sequences* and *depth first sequences* are constructed in a similar manner and are based on tree traversal mechanisms.

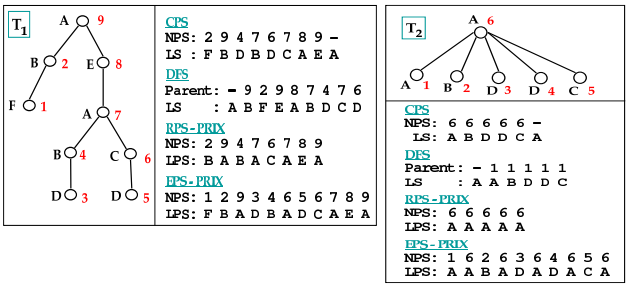

Figure 1: Example Database

Consolidated Prüfer Sequence (CPS): This representation is inspired from classic Prüfer sequences proposed by Heinz Prüfer in [27]. CPS of a tree T consists of two sequences – *Numbered Prüfer Sequence* NPS_T and *Label Sequence* LS_T (see Table 1). They are constructed by doing a post-order traversal that tags each node in the tree with a unique traversal number. NPS is then constructed iteratively by removing the node with the smallest traversal number and by appending its parent node number to the already constructed partial sequence. LS is constructed similarly but by taking the *node labels* of the deleted nodes instead of their parent node numbers. Both NPS and LS convey completely different but complementary information – NPS that is constructed from unique post-order traversal numbers gives the tree structure and LS gives the labels for each tree node. CPS representation thus provides a bijection between rooted ordered trees and sequences, as captured in Lemma 3.1. Note that *each entry in the CPS corresponds to an edge* in the tree. For simplicity, we refer to CPS as prüfer sequence in the rest of the paper.

LEMMA 3.1. *CPS=(NPS, LS) uniquely represents a rooted, ordered tree.*

Few example sequences are shown in Figure 1. Adjacent to each tree node its post-order traversal number and its label are shown. The figure also shows *Regular Prüfer Sequence* (RPS) and *Extended Prüfer Sequence* (EPS) representations proposed in *PRIX* [26]. Note that unlike CPS's label sequence, RPS stores a Labeled Prüfer Sequence (LPS) that is constructed by taking the *parent node labels* of the deleted nodes. Since both the sequences in RPS (i.e., NPS and LPS) include only the information of parent nodes, RPS *can not* represent the leaf nodes. In order to incorporate them, PRIX extends each leaf with a dummy node and then constructs NPS and LPS on top of the extended tree, resulting in an Extended Prüfer Sequence (EPS). Though the extended sequences represent the leaf nodes, they can potentially be longer in size (T_2 in Figure 1). The RPS takes as much space as CPS but it can not represent the leaf nodes and the EPS incorporates the leaf nodes but can result in longer sequences – CPS is thus more *space efficient* when compared to both RPS and EPS.

Depth First Sequence (DFS): Depth first sequences are constructed in a similar manner to Prüfer sequences except that they are based on pre-order traversal instead of post-order. The *Parent* sequence is similar to NPS and stores the pre-order numbers of the parent nodes. Every property that holds for CPS also holds for depth first sequences. For example, DFS also uniquely represents a rooted, ordered tree. Therefore, any algorithm that is designed for CPS can be adopted for depth first sequences. The DFS representation for example trees T_1 and T_2 is shown in Figure 1.

3.2 Tree Matching Algorithm

The tree matching algorithm operates on the sequential representations of a database tree T and a given query Q and enumerates the set of matches of Q in T. Without loss of generality, let the number of nodes in Q and T is m and n, respectively. Tree matching is done in three phases – *checking for subsequence*, *subsequence matching*, and *structure matching*.

3.2.1 Checking for Subsequence

DEFINITION: 3.1. *A sequence* $X = \{x_1,...,x_m\}$ *is said to be a subsequence of another sequence* $Y = \{y_1,...,y_n\}$ *if* $x_j = y_{i_j}$, $1 \le j \le m$, $1 \le i_j \le n$, *and* $i_1 < i_2 < ... < i_m$.

THEOREM 3.1. *Consider a tree* T *and a twig query* Q *with their label sequences* LS_T *and* LS_Q, *respectively. If* Q *is a subtree of* T *then* LS_Q *is a subsequence of* LS_T.

PROOF. If Q is a subtree of T then for each node $v_i \in Q$ there exists a node $u_i \in T$ such that each edge (v_i, v_j) in Q corresponds to an ancestor-descendant or a parent-child relationship between u_i and u_j in T. Moreover, the order among the child nodes of v_i in Q is preserved from T. Therefore, the order in which v_i's are deleted while constructing LS_Q is same as the order in which u_i's are deleted when constructing LS_T. The label sequence of twig query Q is thus a subsequence of the label sequence of tree T. □

Theorem 3.1 provides a necessary but not sufficient condition for twig matching. This theorem can be leveraged to recast the problem of subtree isomorphism into the problem of subsequence matching. We thus evaluate the necessary condition for twig match i.e., we check whether or not the sequence LS_Q is a subsequence of LS_T.

PROPERTY 3.1. *If a label sequence LS_Q is a subsequence of another label sequence LS_T then LS_Q is the longest common subsequence of LS_Q and LS_T.*

From the above property, in order to check if LS_Q is a subsequence of LS_T, it is sufficient to check their longest common subsequence (LCS). Finding a longest common subsequence is a well-addressed problem in the literature. A large number of algorithms have been proposed for this purpose [3, 15, 28]. We employ a traditional dynamic programming based approach where the LCS length between two input strings $LS_Q[1..m]$ and $LS_T[1..n]$ is computed by finding the LCS lengths for all possible prefix combinations of LS_Q and LS_T. Computed LCS lengths are stored in a matrix and are used later in finding the LCS length for longer prefixes – dynamic programming. Equation 1 gives the recurrence relation for extending the LCS length for each prefix pair ($LS_Q[1..i]$, $LS_T[1..j]$) [28].

$$R[i,j] = \begin{cases} 0, & if\ i=0, j=0 \\ R[i-1,j-1]+1, & if\ LS_P[i] = LS_T[j] \\ max(R[i-1,j], R[i,j-1]), & if\ LS_P[i] \neq LS_T[j] \end{cases} \quad (1)$$

An entry $R[i,j]$ gives the LCS length for a prefix pair ($LS_Q[1..i]$, $LS_T[1..j]$). The bottom-right corner entry $R[m,n]$ gives the overall LCS length. If that is different from m then Q is not a subtree of T, from Property 3.1 and Theorem 3.1.
Complexity Analysis: The recurrence in Equation 1 scans every entry in R exactly once. Therefore the run time complexity of this algorithm is $\Theta(mn)$.

3.2.2 Subsequence Matching

Given that LS_Q is a subsequence of LS_T from previous step, Algorithm 1 enumerates the set of all subsequence matches of LS_Q in LS_T. An initially empty subsequence match is progressively constructed by recursing on the R-matrix. The match is extended whenever the label sequences match at positions given by $Qind$ and $Tind$ (line 1). If they do then the position of the match in T i.e., $Tind$ is recorded in line 2 and the backtracking is continued towards $R[1,1]$ (lines 6-9). Since R is processed from the bottom-right corner to the top-left corner, subsequence match is established *from right to left*. Furthermore, each subsequence match is generated exactly once in the process.

Algorithm 1 Subsequence Matching Algorithm

```
naiveBacktrack (Qind, Tind)
Input: LS_T, LS_Q, R-matrix
 1: if LS_Q[Qind] = LS_T[Tind] then
 2:    record the match for LS_Q[Qind]
 3:    naiveBacktrack ( Qind-1, Tind-1 )
 4:    naiveBacktrack ( Qind, Tind-1 )
 5: else
 6:    if R[Qind-1, Tind] > R[Qind, Tind-1] then
 7:       naiveBacktrack ( Qind-1, Tind )
 8:    else
 9:       naiveBacktrack ( Qind, Tind-1 )
```

Each of the resulting subsequence match (SM) is denoted by the list of positions at which the match is found – (i_1,...,i_m) i.e., $LS_Q[k]= LS_T[i_k]$, $1 \leq k \leq m$. Alternatively, SM is represented as a sequenture that is formed by the entries of CPS(T) taken from the matching positions. More formally,

$$SM = ((NPS_T[i_1], LS_T[i_1], i_1)...(NPS_T[i_m], LS_T[i_m], i_m)) \quad (2)$$

Note that, the sequenture representation in Equation 2 is an *alternative representation* to the sequence of matching positions (i_1, ..., i_m).

Recall that Theorem 3.1 is only a necessary condition and hence a subsequence match found in this step may not match the given twig structurally. Such false positive subsequence matches are pruned in the next step, structure matching.
Complexity Analysis: Algorithm 1 is clearly exponential as the potential number of subsequence matches enumerated by it is exponential. We thus analyze its complexity in terms of the *total number of recursive calls* made. Consider a worst case scenario where each node in the tree and the twig has the same label. In this case, only the recursions in lines 3 and 4 are executed. Say that $a_{m,n}$ is the number of recursions made by Algorithm 1 in the worst case and $b_{m,n}$ is the number of matches in the worst case, where m and n are the input parameters $Qind$ and $Tind$, respectively. The recurrences to derive the values of $a_{m,n}$ and $b_{m,n}$ are shown in Equation 3 and Equation 4. The exponential nature of $a_{m,n}$ is evident from the second condition in Equation 3. The recurrence of $b_{m,n}$ has a nice closed form $\binom{n}{n-m}$. As an example, for a twig of size 5 and a tree of size 10, the algorithm makes $a_{5,10}$=1,275 number of recursions to find $b_{5,10}$=252 subsequence matches.

$$a_{m,n} = \begin{cases} 1 + a_{m-1,n-1} + a_{m,n-1}, & if\ n > m \\ 1 + 2 * a_{n-1,n-1}, & if\ n = m \\ 1 + 2 * n, & if\ m = 1 \end{cases} \quad (3)$$

$$b_{m,n} = \begin{cases} \sum_{i=1}^{n-m+1} b_{m-1,n-i}, & if\ n > m \\ 1, & if\ n = m \\ n, & if\ m = 1 \end{cases} \quad (4)$$

3.2.3 Structure Matching

The subsequence matching algorithm considers only the label information and ignores the structural information. There can exist multiple twig patterns which differ in structure but with the same label sequence. Recall that Theorem 3.1 provides only a necessary condition for twig matching. Our *structure matching* algorithm prunes these false positive subsequences by considering the structure given by NPS.

DEFINITION: 3.2. **Structure Agreement:** *Consider two sequentures, derived from two trees T_1 and T_2, S_1 = ((A_1, B_1) ... (A_m, B_m)) and S_2 = ((C_1, D_1) ... (C_m, D_m)), where A_i's and C_i's define the structure; B_i's and D_i's provide the labels. Both S_1 and S_2 are said to* **agree on structure at position i** *if and only if the following three conditions hold:*

i) $1 \leq i \leq m$,
ii) B_i *is equal to* D_i,
iii) If A_i *is the parent of* B_i *in* T_1 *then* C_i *is the parent*

of D_i or the nearest ancestor of C_i that is in S_2 must agree on structure with S_1 at position A_i [1].

THEOREM 3.2. *A twig query Q is a subtree of T if and only if its Prüfer sequence CPS(Q) and its subsequence match SM at locations $(i_1, ..., i_m)$ in T agree on the structure at all positions k, $\forall k, 1 \leq k \leq m$.*

PROOF. **IF**: From the definition of a subsequence, the first two conditions of Definition 3.2 trivially hold true for CPS(Q) and SM. Consider an entry in the twig's prüfer sequence $(NPS_Q[k], LS_Q[k])$ and its corresponding entry $(NPS_T[i_k], LS_T[i_k])$ in its SM. Recall that each entry in a Prüfer sequence is an edge. $NPS_Q[k]$ is thus the parent of the query node at position k. Since Q is an embedded subtree of T, this edge corresponds to a parent-child relation in SM (i.e., $NPS_T[i_k]$=$NPS_Q[k]$ – first part of condition *iii*) or an ancestor-descendant relation SM (second part of condition *iii*).

ONLY IF: An isomorphism between Q and its subsequence match is a bijective label-preserving function f that maps every vertex v_k in Q to a vertex $f(v_k)$ in T in such a way that the adjacencies are preserved. In other words, for every node v_j that is adjacent to v_k there is a node $f(v_j)$ that is adjacent to $f(v_k)$.

Now consider a function $g : Q \rightarrow SM$ such that g(v_k) = u_{i_k}, v_k's and u_k's are vertices in Q and T, respectively. Furthermore, assume that g satisfies all the conditions in Definition 3.2 at each k. First two conditions infer that g is a label-preserving function. The third condition infers that g maps every edge in Q to an parent-child or ancestor-descendant relation in T. Since the node relationships are determined by edges, we can deduce that g preserves the adjacencies among vertices in Q. Therefore, g defines an isomorphism between Q and the subtree formed by its subsequence match in T. □

From the above theorem, it is sufficient to check for the structure agreement between Q and its subsequence match SM at all positions in order to find whether Q is a subtree of T or not. The structure matching algorithm based on this theorem is shown in Algorithm 2.

Algorithm 2 Subtree matching

Input: CPS(Q), CPS(T), SM=$(i_1, .., i_m)$
Output: *mapping*: positions at which Q matches to a subtree in T

```
1: mapping[m] ← i_m
2: for k = m − 1 to 1 do
3:     p_q ← NPS_Q[k]
4:     p_t ← NPS_T[i_k]
5:     if mapping[p_q] is equal to p_t or is an ancestor of p_t in T
       then
6:         mapping[k] ← i_k
7:     else
8:         Report that Q is not an embedded subtree of T
9: Report that Q is an embedded subtree of T
```

Algorithm 2 iteratively process the nodes in Q and stores the structure match found so far in *mapping* array. The structure agreement check at a node is performed only after processing all of its ancestors. The nodes in the twig are thus processed from right-to-left i.e., from root to leaves (line 2). Since the root node does not have any ancestors, it is mapped without performing any checks (line 1). As soon as the check at position k is successful, the matching node in T at position i_k is recorded in *mapping* array (line 6). To perform such a check, the algorithm considers the parent node numbers from k^{th} entry in both the prüfer sequence of Q and its subsequence match SM (lines 3-4). The algorithm checks if the node mapped to p_q is same as p_t or is an ancestor of p_t (line 5). By doing so, the algorithm makes sure that a parent-child relation in Q is translated into an ancestor-descendant relation in T. If the check fails at any position, it can be concluded that Q is not a subtree of T. Since SM is a label match to Q, Algorithm 2 does not check for labels. Note that, $mapping[p_q]$ in line 5 will always have a value as the structure match is established from right to left.

Complexity Analysis: Algorithm 2 accesses each node in the twig exactly once (line 2). This scan takes $\Theta(m)$ time. The structure agreement check at each position traverses the path from i_k to $mapping[p_q]$ (line 5). In the best case where $p_t = mapping[p_q]$, the traversal takes $\Theta(1)$ time. In the worst case, p_q is mapped to the root of T in which case the traversal time is in the order of the tree's *depth*. The complexity of line 5 in such cases is $O(n)$ because the maximum depth of a tree is equal to the number of nodes. Therefore, Algorithm 2 has a *best case run time complexity* of $\Omega(m)$ (in case of induced subtrees [2]) and a *worst case complexity* of $O(mn)$. The constant factor involved is usually small because in the worst case scenario (a single long path), if the check at a node A traverses till the root then the checks for all A's descendants will at most traverse up to A and never hence reaches the root node.

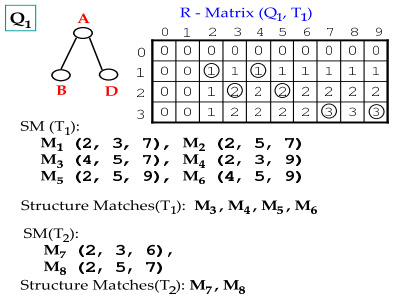

Figure 2: Example: *R*-Matrix and Matches

Example: Figure 2 shows the 6 matches of the twig Q_1 in trees T_1 and T_2, which are shown in Figure 1. Every match represents a sequenture. For example, the sequenture for M_1 is $((B,9,2)\ (D,4,3)\ (A,8,7))$ and for M_3 it is $((B,7,4)\ (D,5,6)\ (A,8,7))$. Algorithm 2 detects both M_1 and M_2 as false positives because they fail the structure agreement check at position 1. For example, for M_1, (i_1,i_2,i_3)=(2,3,7). The structure matching algorithm maps the root node A (node# 3) in Q to node 7 in T_1 and maps D (node# 2) in Q_1 to node 3 in T_1. Now consider the check at position 1. The parent of node# 1 (B) in Q_1 is 3 i.e., p_q=3 and p_t=$NPS_{T_1}[i_1]$=9. The mapped node for p_q, $mapping[p_q]$=7, is not same as p_t and it is not an ancestor of p_t. Therefore, the check at position 1 fails for M_1 and hence it is not a structure match. Similarly, M_2 does

[1] For an *induced* subtree match, the condition (*iii*) checks only the first part i.e., parent-child relationship.

[2] Twig queries with no wildcards result in induced subtree matching.

not agree on structure at position 1. In T_2, Both the subsequence matches, M_7 and M_8, can be shown to be actual twig matches for Q_1.

As noted earlier, the ancestor check in line 5 can traverse till the root node, in worst case. Approaches such as scoping [32] and range encoding [23, 29] are known for making fast ancestor checks. They take $O(1)$ time as we just need to check if $p_t \in Scope(mapping[p_q])$ or not. However, this seemingly accurate approach cannot find all the embeddings of Q and it can potentially generate false positive structure matches. For example, consider the twig query that is matched with the nodes 2, 6, 8, and 9 in T_1. The scope-based approach incorrectly flags the subsequence match (4, 6, 8, 9) as a valid structure match. Therefore the traversal in line 5 is mandatory for the correctness of Algorithm 2.

3.3 Indexing Method

The tree matching algorithm in previous section is defined for a given database tree. In order to find the matches in the database D, every tree in D has to be processed – an inefficient approach. Instead, we make use of a simple *inverted index* on distinct labels in D to restrict the number of trees for which the matching is made. Against each unique label l in D, we store a list of tree identifiers (S_l) in which l occurs. The tree matching algorithm is applied on those trees in which every query label occurs i.e., $\bigcap S_l$, $\forall\ l \in Q$. As a heuristic, one can simply choose a list with the smallest size to avoid the potentially expensive intersection step. Such an index structure is *independent* of the tree matching algorithm as the index is probed before the algorithm is applied. On a contrary, most of the state-of-the-art methods leverage the index structure during the subsequence matching phase.

Index Maintenance: In any case of insert, delete, or update the changes to be made to the index are minimal as only the lists of modified nodes are affected. However, in the case where a tree is updated by adding or deleting the nodes, few NPS entries might have to be changed to reflect the new post-order traversal numbers. In the worst case, when the change is at the left-most-leaf, the entire NPS needs to be updated. To reduce the impact of such cases, a batched *lazy-update* approach can be employed where a set of updates are grouped and applied on the database as a batch.

Index size: The size of our index structure is completely dependent on the distribution of labels in the database. If the database trees are highly associative then the size of each list is approximately equal to the database size. However, we observed that most of the real datasets exhibit a highly skewed distribution i.e., very large number of labels have small lists and very few labels have large lists. To reduce the index size, we propose an *α-infrequent index*.

DEFINITION: 3.3. *An index is called α-infrequent if it stores the tree identifier lists for only those labels which appear in less than a fraction of α trees in the database.*

For example, a 0.8-infrequent index does not store lists for labels which appear in more than 80% of trees. A 1-infrequent index maintains lists for every label in the database and a 0-infrequent index is an empty index with no lists. If the list for a particular label of interest is not available then we consider the entire set of database trees. Furthermore, our α-infrequent index can easily be made *out-of-core* by storing few large lists on the disk.

3.4 Optimizations

As shown in section 3.2.2, the number of recursions can be quite high when compared to the number of subsequence matches. We now present a series of inter-linked filtering and pruning optimizations which speed up the tree matching process by reducing the unwanted overhead.

3.4.1 Label Filtering (LF)

This optimization relies on the distribution of labels in a database tree (T) over the nodes in the given twig query Q. It is based on the observation that the number of distinct labels in T is usually a lot higher than the number in Q. Consider a partitioning of labels in T into two mutually exclusive subsets, V_1 and V_2, such that $V_1 = \{l | l \in T,\ l \in Q\}$ and $V_2 = \{l | l \in T\}$ - V_1. The values in R-matrix columns corresponding to the labels in V_2 simply carry the values of LCS values from one column to another and thus do not contribute in building the subsequence match and hence they can safely be excluded. In other words, LS_T can be pruned from the labels in V_2 before constructing the R-matrix. For example in Figure 2, the columns corresponding to labels C, E, and F are removed. This pruning process is called as *Label Filtering*. The advantages of label filtering are two-fold. *First*, it reduces the size of R-matrices thereby making them fit in few cache lines. *Second*, it reduces the recursion overhead in Algorithm 1 as the irrelevant entries in R-matrix are now discarded.

3.4.2 Dominant Match Processing (DM)

In cases where the set of all distinct labels in both the query tree and the data base tree are almost same, the benefits from label filtering would be marginal. This optimization is designed to further reduce the number of recursions by limiting the backtracking to a very few entries, called as dominant matches, in the R-matrix.

Based on the values, the entries in R can be partitioned into classes: $C_k = \{(i,\ j) |\ R[i,j] = k\}$, $1 \leq k \leq m$. For the example matrix in Figure 2, C_3 is $\{(3,7), (3,8), (3,9)\}$. Algorithm 1 accesses the matrix elements in the decreasing order of their class because the match is constructed from right to left. Backtracking from $R[i,j]$ proceeds into two regions of the matrix – R_1: from $R[1,1]$ to $R[i\text{-}1, j\text{-}1]$ and R_2: from $R[1,1]$ to $R[i, j\text{-}1]$. Therefore, while extending the match from $R[i,j] \in C_k$ only those elements from these two regions which belong to C_{k-1} need to be considered. For example in Figure 2, recursive calls from $R[3,6]$ need not be made for elements with value same or more than $R[3,6]$ (e.g, $R[2,4]$).

Furthermore, only few elements of R, for which the second condition in Equation 1 holds true, contribute towards the subsequence match. These entries are identified at line 2 in Algorithm 1. Other elements, which satisfy the third condition in Equation 1, simply carry forward the LCS length through recursive calls. Say $R[i,j]$ and $R[k,l]$ ($i < k, j < l$ are two entries at which the LCS length is increased. Furthermore, assume that $\nexists x$ such that $j < x < l$, and $LS_Q[k] = LS_T[x]$. In Algorithm 1, backtracking from $R[k,l]$ can directly jump to $R[i,j]$ as the intermediate cells simple carry the LCS value from $R[i,j]$ to $R[k,l]$. We refer to the cells $R[i,j]$ and $R[k,l]$ as *dominant matches*. In Figure 2, dominant matches for Q_1 in T_1 are encircled. Entries $R[2,6]$, $R[3,5]$, and $R[3,6]$ can be ignored from processing as they just carry the LCS value from $R[3,7]$ to $R[2,5]$.

Considering only the dominant matches imply that elements need to be considered in the decreasing order of their *row* & *column* indices (i.e., region R_1). The decreasing order is because the match is constructed from right to left. Therefore, backtracking from $R[i,j] \in C_k$ can now be limited to just dominant matches in R_1 which belong to C_{k-1}. Such a stringent condition drastically reduces the number of recursions while matching the subsequence (see Section 3.5 for analysis). Furthermore, no special data structure is needed to implement this optimization – dominant matches can just be stored as negative numbers in R.

In Figure 2, backtracking from $R[2,5] \in C_2$ needs to consider both $R[1,4]$ and $R[1,2]$ from C_1 in that order whereas processing from $R[2,3] \in C_2$ needs to consider a single element $R[1,2]$ from C_1.

3.4.3 Simultaneous Subsequence and Structure Matching

Both the optimizations LF and DM are targeted at the subsequence matching phase. They do not try to reduce the number of false positives fed into the structure matching phase. For Q_1 and T_1 in Figure 2, only 4 of the 6 subsequence matches are actual subtree matches. These false positives are identified and filtered only in the structure matching phase. They, when in large number, add a significant filtering overhead and may hence hinder the performance. This overhead is alleviated by detecting the false positives as early as possible. It is accomplished by integrating the structure matching into the process of subsequence matching i.e., both are done simultaneously. Such unification is feasible due to two reasons: *First*, both the matching algorithms operate on the the nodes in the same order, right-to-left. *Second*, a structure agreement check for a node at position i needs only its ancestors, for which the structure match is already established. As soon as a label match at i^{th} position (i.e., for $LS_Q[i]$) is found by the subsequence matching algorithm, the structure agreement check at that position is conducted by the structure matching algorithm [3]. By pushing the structure constraints deep into the process of subsequence matching the unified approach yields a better performance [24].

3.5 Putting it all together

Our complete unified tree matching algorithm to find the set of all twig matches for Q in T is shown in Algorithm 3. T's label sequence is first filtered from labels which are not in Q, using LF. Equation 1 is then used to construct the R-matrix. While constructing the matrix, the dominant matches are marked whenever both the label sequences match as per the second condition in Equation 1. The function *processLCS* is invoked to enumerate the set of *all* twig matches.

For a given position $Qind$ in the twig, *processLCS* tries to find a label match in T (line 3). As soon as a match is found, Algorithm 2 is executed at the matched location by the function $isInAgreement()$ (line 5). The resulting structure match is recorded in SM at line 6. If the user is aware of the exact document structure then the user can provide the exact level difference between a node and its parent node. Such *level-wise constraints*, which can easily be incorporated in to $isInAgreement()$, extend the capabilities of standard wildcards ('//' and '*') and improve the matching process significantly. The twig match is recursively extended by backtracking in line 7 and is reported in line 2.

[3]In case of DFS, either the backtracking has to start from the top left corner of R or the matrix has to be constructed on the reversed label sequences.

Algorithm 3 The unified subtree matching algorithm

Input: A database tree T and a twig query Q
labelFilter (T, Q) {T contains the filtered sequence}
$R \leftarrow$ *computeLcsMatrix* (T, Q)
if $R[m,n] != m$ **then**
  Report that Q is not a subtree of T
$SM \leftarrow null$
processLCS (m, n, m)

Function:
processLCS ($Qind$, $Tind$, $matchLen$)
1: **if** $matchLen = 0$ **then**
2: Report SM as the twig match
3: **for** $i = Tind$ to 1 **do**
4: **if** $R[Qind][i]$ is dominant & $R[Qind][Tind] = matchLen$ **then**
5: **if** isInAgreement(CPS(Q), SM, $Qind$) **then**
6: $SM[Qind] \leftarrow CPS_T[Tind]$
7: *processLCS* ($Qind$-1, $Tind$-1, $matchLen$-1)

Complexity Analysis: As mentioned before since the enumerated twig matches are exponential in number, the overall time complexity of the algorithm would be exponential. Similar to the section 3.2.2, we now analyze the number of recursions made by our unified algorithm in a worst case scenario where all the tree nodes have the same label. Similar to $a_{m,n}$, define $c_{m,n}$ to be the number of recursions made by the unified algorithm in the worst case. Since only the dominant matches from region R_1 are processed, Equation 5 involves just one term $c_{m-1,n-1}$. $c_{m,n}$ has a closed form of $\binom{n+1}{n-m+1}$. Clearly, label filtering and dominant match processing significantly reduces the the number of recursions. For example, while finding matches for a twig of size 5 in a tree of size 10, the unified algorithm makes 462 recursive calls where as the naive algorithm makes 1,275 number of recursions.

$$c_{m,n} = \begin{cases} 1 + \sum_{i=1}^{n-m+1}(1 + c_{m-1,n-1}), & if\ n > m \\ n, & if\ n = m \vee m = 1 \end{cases} \quad (5)$$

In summary, we leverage a dynamic programming based approach instead of a traditional index-based approach for finding the set of all subsequence matches. Our novel unified matching algorithm and employed optimizations are expected to significantly reduce the query processing time. The nature of our approach and its reliance on sequential encoding and matrix or array based data structures reflects a cache conscious design with small memory footprints. Furthermore, the avoidance of pointer-based data-structures ensures that the instruction level parallelism of the approach will not be affected. Finally, our algorithm is trivially adaptable in a data-parallel setting. Essentially the database of trees is partitioned and each processor computes the matches from one partition and the results are combined at the end. Such designs are quite important and relevant when placed in the context of emerging multi-core architectures (see Section 4.3).

4. RESULTS

We now evaluate the performance of our algorithm LCS-TRIM against PRIX and three different *TwigStack* algorithms: TwigStack (uses no index), XBTwigStack (uses XB tree), and XRTwigStack (uses XR tree – TSGeneric$^+$ in [17]). We have obtained PRIX from its open source distribution[4] and TwigStack algorithms from the authors of [17]. All the experiments, unless otherwise noted were performed on a system with dual AMD 250 Opteron processors and 8 GB of main memory. We have found that the performance of LCS-TRIM using DFS is quite similar to the results obtained using Prüfer sequences, the ones we present here.

Dataset	# of Trees	Max Depth
Swissprot	50,000	5
Treebank	52,851	36
DBLP	328,858	6
Cslogs	59,691	85
NLM	450K-1M	8

Table 4: Datasets

Datasets: We consider five different data sets – Swissprot, Treebank, DBLP, Cslogs, and NLM (see Table 4). These five datasets are derived from five different application domains and have different characteristics. Swissprot (curated protein), Treebank (syntactic structure of English sentences), and DBLP data sets are obtained from the University of Washington XML data repository[5]. Cslogs data set, that represents the website access patterns, is obtained from Dr. Zaki's website [6]. Finally, NLM data set[7] houses abstracts from MEDLINE and other life science journals for biomedical articles. These XML data sets are parsed using a SAX parser [8] and the output is then converted into sequences using methods in Section 3.1.

Query workload: Table 2 and Table 3 shows the query workload we use in the evaluation. The queries are carefully chosen such that each one has different characteristics – large and small queries, deep and bushy queries, low and highly selective queries.

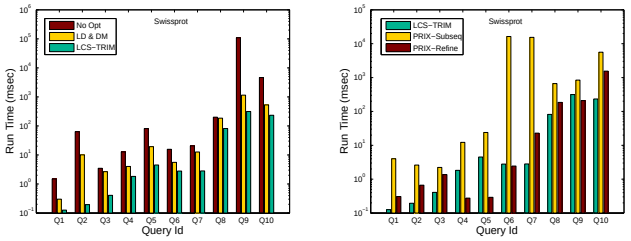

Figure 3: Swissprot (a) Effect of optimizations (b) PRIX run time split

Effect of optimizations: The performance of our algorithm with and without optimizations on the Swissprot query workload is shown in Figure 3a. The basic algorithm with no optimizations is labeled as *No Opt* and the effect of subsequence matching optimizations is shown as *LF & DM*. Note that our *No Opt* algorithm is similar in structure, and expected to exhibit comparable performance, to the methods proposed by Zezula *et al* [35] because one of the main differences between LCS-TRIM and these methods are the use of our optimizations (see Section 2 for all the differences). The recursion overhead is greatly reduced by both LF and DM. They, for example on Q_9, showed a 94-fold speedup that is due to two reasons – the reduction in the average R-matrix size from 28KB to 8KB and the reduction in the number of recursive calls to Algorithm 1 from 16 billion to 5 million. The optimizations targeting structure matching in conjunction with subsequence matching, denoted LSC-TRIM, are very effective across the board. For example, looking at the query Q_9 again, we find that of the 3.2 million subsequence matches only 17,836 of them are actual structure matches. A large majority of these false-positives are filtered out by our unifying optimization in the early stages resulting in an overall speedup of 345 for this query against *No Opt*.

4.1 Comparison with PRIX

Figures 4 and 5 demonstrate the performance differences between LCS-TRIM and PRIX. LCS-TRIM achieves a significant speedup over PRIX in enumerating all the twig matches – on average, up to 3 orders of magnitude speedup on DBLP and up to 2 orders on other data sets.

This huge disparity in run times is due to various reasons. *First*, the subsequence matching step in PRIX is very expensive. Too many range queries issued during the process increase the number of accesses to the index and the large B^+*-tree* index structures (see Figures 5 and 6) make those accesses very costly. *Second*, once we factor in the need to use the *EPindex* (index constructed over extended prüfer sequences) for value-based queries the number of accesses to the index increases dramatically. For query Q_4, there are 2,041 subsequence matches requiring 690 accesses to the index. PRIX then further refines these 2,041 matches using a series of post-processing steps to determine that all but only 9 of them are false positives. In contrast, LCS-TRIM filters these the false positives very early by combining the subsequence and structure matching phases. *Third*, by way of comparison our approach relies on an inverted tree index to identify candidate trees and for each candidate tree we rely on the processing of small R-matrices – which can often fit in a few cache lines – leading to excellent locality. Drilling down on this disparity even further in Figure 3b we see that in all of the queries the base subsequence match step in PRIX is already more expensive than our method. Factoring in the time for refinements (for PRIX to generate the correct results) adds to the disparity. For example, consider the fact that subsequence matching in PRIX for Q_7 took 15,300$msec$ and refinements amount to an additional 29$msec$. For the same query our approach has taken just about 2$msec$. The refinement time for queries Q_4 and Q_5 is small due the fact that those queries are small in size and also have a small number of matches. Our overall query processing time is usually *less than* the time that PRIX spent in refining the subsequence matches.

Effect of pivots and gap constraints: PRIX relies on user-provided pivots and gap constraints in order to reduce the number of accesses to the index, while finding the subsequences. PRIX leverages *pivots* in performing a bi-directional subsequence matching that is done using two separate indexes *LIndex*, *RIndex* [26]. The performance of PRIX is highly dependent on the pivots chosen. Pivots pointing to low selective nodes may actually increase the

[4] http://www.cs.arizona.edu/prix/

[5] http://www.cs.washington.edu/research/xmldatasets/

[6] http://www.cs.rpi.edu/~zaki/software/

[7] http://www.ncbi.nlm.nih.gov/entrez/query.fcgi?DB=pubmed

[8] http://www.saxproject.org/

Qid	XPath Expression	Matches
Q_1	//Entry[PFAM[@prim_id=PF00304]][//SIGNAL/Descr]	3
Q_2	//Entry[Org][PFAM[@prim_id=PF00304]][//SIGNAL/Descr]	39
Q_3	//Ref/Author="Moss J"	38
Q_4	//Entry [Species="Glycine max"][Organe="Chloroplast"] [Org="Glycine"]	9
Q_5	//Features/DOMAIN[from="165"][to="171"] [Descr="POLY-PRO"]	1
Q_6	//Features[/*/from][//"171"][//"POLY-PRO"]	14
Q_7	//Features[//"165"][/*/to][//"POLY-PRO"]	51
Q_8	//Entry/Features/DOMAIN	46,566
Q_9	//Entry[Org="Eukaryota"][Org="Metazoa"] [Org="Chordata"] [Org="Craniata"]	17,836
Q_{10}	//Entry[/*/"Eukaryota"][/*/"Metazoa"][/*/"Chordata"] [/*/"Craniata"]	17,836

Qid	XPath Expression	Matches
Q_{25}	//1/3[//545]	26
Q_{26}	//2500[7695][2501]	72
Q_{27}	//1155[//4525][/ * /2613/5888]	82
Q_{28}	//1155[//5996//7834][2586]	85
Q_{29}	//1155[2586][//5996]	3,742
Q_{30}	//1/271/272/273//276//278/ * /281//284//287/1552]	78
Q_{31}	//5191[8650][8686][8685][8684][9400]-[8680]	6
Qid	XPath Expression (NLM)	
Q_{32}	//MedlineCitation[//Year="1999"][/Article[//Journal][//Author][//Author]]	
Q_{33}	//MedlineCitation[/DataCreated//1999][/DataRevised/2000]	
Q_{34}	//MedlineCitation[/Article//Volume="153"]	

Table 2: XPath queries for Swissprot, Cslogs, and NLM data sets

Qid	XPath Expression	Matches
Q_{11}	//inproceedings/*/"Antonin Guttman"	2
Q_{12}	//article/author="Antonin Guttman"	3
Q_{13}	//phdthesis[year][series][number]	1
Q_{14}	//phdthesis[year][number]	3
Q_{15}	//inproceedings/author="E. F. Codd"	33
Q_{16}	//book[//AA93][//AABM82][//AB87a][//AB87b][//AB88][//AB91][//ABD+89]	1
Q_{17}	//article[year=1999]	7,408

Qid	XPath Expression	Matches
Q_{18}	//EMPTY/*/LS_OR_JJ	2
Q_{19}	//S/*/RB_OR_JJ	8
Q_{20}	//S/SBARQ-1	34
Q_{21}	//NP/ADJP/IN_OR_RB	1
Q_{22}	//S[PRT][NP]	2
Q_{23}	//EMPTY//X/VP/PP/NP/S/VP/VP/NP	107
Q_{24}	//EMPTY[/*/NP][/*/X[//VBN][//WRB][//S[/*/_NONE_][//VBG]]]	12

Table 3: XPath queries for DBLP and Treebank data sets

number of accesses to the index and thereby may hinder the performance. For example, in queries Q_{18} and Q_{19}, the nodes EMPTY and S have low selectivity with 49,416 and 50,726 matches, respectively. When they are chosen as pivots, the run time for Q_{18} and Q_{19} increased to $0.15sec$ and $0.8sec$ – a significant slowdown when compared to the best pivot choice ($0.3msec$ and $0.6msec$ as shown in Figure 4). All the reported results for PRIX are based on the best choice. Unfortunately, providing the best pivot value can be challenging for the user. The database engine can of course provide some hints based on selectivity estimation but they may not be optimal. Our methods on the other hand do not depend on such choices. In fact, even when we process the entire data set the retrieval times are not very high – $0.014sec$, $0.018sec$ for Q_{18} and Q_{19}, respectively.

Gap constraints are specified by the user based on the distance between two elements in the database sequence. For example in Q_{22}, occurrences of tag PRT in the database is scattered over several documents. Only 2 of these documents have S as PRT's parent while in others, S is an ancestor. PRIX, in this case, relies on gap constraints to restrict the search to parent-child axis. With sub-optimal gap constraints, the total number of index page transfers for Q_{22} have increased from 1,502 to 79,177, reflecting a significant increase in the number of index accesses. Furthermore, inaccurate gap constraints can lead to inaccurate results. In order to provide accurate gap constraints, the user not only needs to know the database tree structure but also its *internal* representation. Even though a level-wise constraint is similar to a gap constraint, the former depends on a high-level document structure where as the latter depends on internal representation. We now consider the workload characteristics in evaluating the performance.

Effect of query size: The number of nodes in the query tree directly affects the size of R-matrix and hence the performance of LCS-TRIM. In our workload, small sized queries are Q_3, Q_{11}, Q_{12}, Q_{19}, Q_{20}, and Q_{25} and large queries are Q_5, Q_{16}, Q_{24}, Q_{30}, and Q_{31}. We have observed an average speedup of 60 and 400 on small and large queries, respectively. Smaller queries will have smaller R-matrices, which mostly fit in the L1 cache resulting in very good locality. A higher speedup on large queries is due to the slowdown in PRIX culminating from an increased number of index accesses. Hence the query size does not affect our simple matrix-based approach as much as it affects PRIX.

Effect of recursive structure: We now examine how a twig's recursive structure affects the performance. In order to do that, consider the queries, Q_{23} and Q_{30}, with a deep structure and the queries, Q_4, Q_9, Q_{16}, Q_{24}, and Q_{31}, which are shallow and bushy. Note that the trees in both Swissprot and DBLP data sets have a small depth (see Table 4). The average speedup on deep queries (240 times) is a little more than the speedup on bushy queries (190 times). This difference can again be attributed to a large number of index accesses made by PRIX in case of deep queries. On query Q_{24} alone LCS-TRIM achieved up to 3 orders of magnitude speedup.

Effect of selectivity: The query workload has some highly selective queries (Q_5, Q_{16}, Q_{21}, Q_{22}, and Q_{31}) and some low selective queries (Q_8, Q_9, Q_{17}, Q_{27}, and Q_{29}) with many matches. LCS-TRIM showed up to 2 orders of magnitude speedup over PRIX on both the type of queries. For low selective queries, PRIX spent a lot of time in accessing the index while finding many subsequence matches. The performance of PRIX is very sensitive to the selectivity of individual nodes in the twig. When the subsequence matching starts off with a low selective node, the performance gets severely affected due to increased number of range queries issued to the index. The effect of node selectivity on the performance of our algorithm is relatively small because of the way our inverted index is probed (before the matching process starts) and the way we find the subsequence matches.

Effect of wildcards: Finally, we analyze the effect of wildcards on the query processing times of LCS-TRIM and PRIX. The queries Q_6, Q_7, Q_{10}, Q_{16}, and Q_{24} have a high number of wildcards. As an example, consider the queries from

Swissprot workload. The queries Q_6 and Q_7 are slight modifications of the query Q_5 with 3 wildcards introduced. These simple modifications have increased the run time of PRIX by more than 600 times. Such a drastic increase is due to three reasons. First, wildcards increase the number of true and false positive subsequence matches. Second, false positives are detected only in the later stages of query processing. Finally, the process of filtering out a subsequence is very inefficient in PRIX – makes too many scans on the twig. LCS-TRIM, on the other hand, embeds employs the unified approach (detects the false positives as early as possible) and makes a single pass on the twig to filter it out. Thus the effect of wildcards on the performance of LCS-TRIM is considerably small when compared to the effect on PRIX's performance. Due to these reasons, we have observed an average speedup of $2,500$ on the queries with many wildcards.

Results on the NLM data set: We now test the performance of both the algorithms on large data sets by considering Q_{32}, Q_{33}, and Q_{34}. We used a system with 2.8GHz P4 processor and 1.5GB of main memory for this experiment. Figure 5 shows the performance differences as the data set size (XML file size) is increased from 0.5GB to 2.5GB. An *α-infrequent (α=0.5) index* is used for this experiment.

On all the queries, when the data set size exceeds the main memory size (1.5GB), there is a significant increase in the run time of LCS-TRIM. Note that the graphs may be a bit misleading since the y-axis has a log-scale basis. For example on Q_{34} when going from a data set size of 1.5GB to 2GB, it appears as though LCS-TRIM suffers a significant slowdown while PRIX only suffers a marginal slowdown. However in reality the execution time of LCS-TRIM goes up to 0.1s whereas the PRIX running time goes up from 1.5s to 2.5s. Essentially once we go out of core the disk latency dominates for both the methods and as a result of Amdahl's law the relative speedup of LCS-TRIM (10% CPU utilization) with respect to PRIX is reduced. LCS-TRIM is still at least one order of magnitude better on all the queries and in some 3 orders of magnitude better. Such huge speedup is observed in case of queries with complex structure and many wildcards (e.g., Q_{32}). We are currently investigating ways in which the out-of-core performance can be further improved using strategies based on data partitioning and tiling.

Index size comparison: In PRIX, the index is constructed based on user-provided *list of tags*, which correspond to labels in the query workload. A *B^+-tree* is created for each tag in the list. Therefore the total size of the index grows linearly with the number of tags. Figure 6 shows the index sizes along with the number of tags used, for each data set. The figure also shows the size of 1-infrequent index constructed by LCS-TRIM. The huge difference, between PRIX and LCS-TRIM, in index size is clearly evident – compare the 256MB DBLP index in PRIX with our index of size just 48MB. Furthermore, when a 0.5-infrequent index is created the size has reduced to 35MB. Even for large data sets (last graph in Figure 5), our α-infrequent inverted index is quite small. In contrast, index structures in PRIX are very large and are usually more than double the data set size – at 2.5GB data set the index size is almost 5GB. To build a generic query processing engine, the *tag list* should contain a large number of labels resulting in very large index structures. More importantly, each tid-list in our inverted index is *bounded by the number of trees* in the data set whereas the size of each *B^+-tree* in PRIX is *proportional to the selectivity of labels*, which is usually very high.

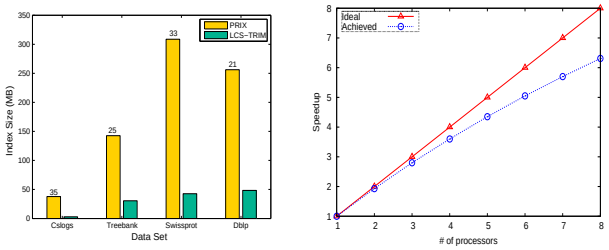

Figure 6: (a) Index Size Comparison (b) Parallel Query Processing

4.2 Comparison with TwigStack

In this section we compare the performance of LCS-Trim with the TwigStack family of algorithms[4, 7, 16, 17]. We should note that TwigStack based algorithms target unordered matches whereas LCS-TRIM targets ordered matches. Later in Section 4.4.1 we discuss some ideas on how to extend LCS-Trim to handle ordered matches. To ensure as fair a comparison as possible, we follow the method employed in PRIX [26], wherein for each query in our workload the number of ordered matches is exactly the same as the number of unordered matches. This is because only one configuration of the twig is present in the data set.

As found in [17], the performance of XR TwigStack is marginally better than the other two algorithms [9] (see Figure 7). Note that the y-axis is shown in log-scale. LCS-TRIM performed significantly better than XR TwigStack on all queries – up to 3 orders of magnitude. The speedup is very high, especially on queries with low selectivity, because the TwigStack algorithms make a large number of scans on huge element lists.

4.3 Parallel Query Processing

LCS-TRIM can be parallelized very easily, making it a viable option for emerging multi-core architectures and for very large distributed databases. It is fairly simple to adopt it in a shared memory or a cluster environment. As a proof of concept, we now present the results from a preliminary analysis using a SMP system with 8 processors and 32 GB of shared memory (see Figure 6). We consider the following query from Cslogs to evaluate the parallel performance.

$$//1155[//5996][//5996][//5996][//5996][//5996]$$

This query has a very high number of matches ($474,716,009$ spread over 608 trees). Each processor in the SMP system picks up a candidate tree from the list returned by the index and finds the twig matches in that tree. We have observed up to 6.3-fold speedup on 8 processors. The distribution of 474 million matches over 608 trees is not uniform. The skew present in the distribution results in load imbalance and therefore a non-linear speedup. We are currently investigating for better load-balancing strategies.

4.4 EXTENSIONS

4.4.1 Unordered matching

[9] We have run this experiment on a Windows PC with a 2.8GHz Intel Pentium 4 processor and 1.5GB of main memory.

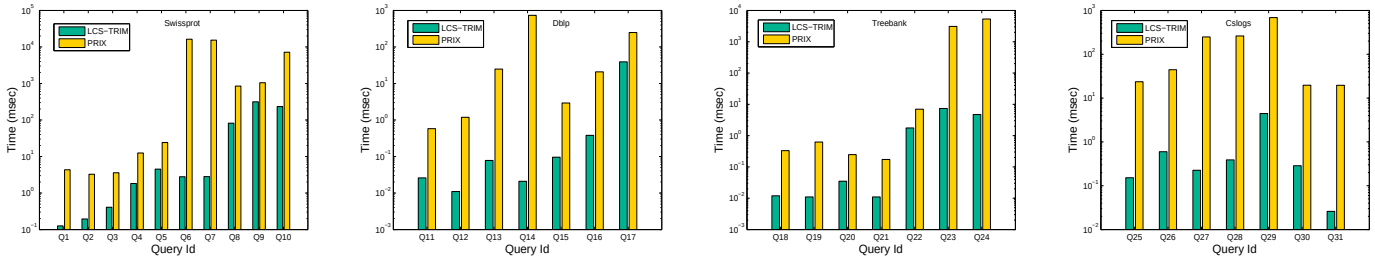

Figure 4: Performance comparison with PRIX on different data sets

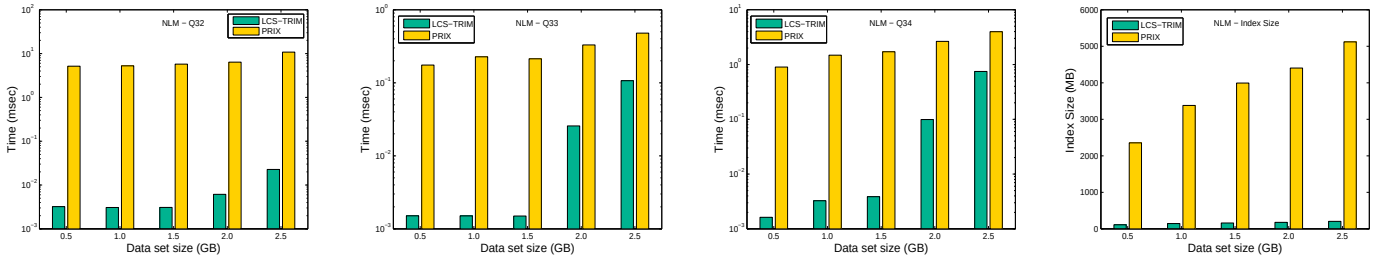

Figure 5: Performance and Index size comparison on NLM data set

The sequencing methods in Section 3.1 are based on the tree traversals and hence they preserve the order among sibling nodes. Therefore, Algorithm 3 can only be applied to process ordered twigs. A typical but naive method to support unordered matches is as follows [26]: First, the set of all configurations (C) of the given twig is found. Second, for each configuration in C, matches are found in each database tree using Algorithm 3. The set C is potentially exponential in the size of the query. However, twig queries are usually small in size and hence processing all the configurations in C may not be very expensive.

There exist several ways to reduce the number configurations which are to be evaluated – by imposing the DTD constraints, by a suitable preprocessing of the data set, and by imposing an artificial ordering on node labels of both the database tree and the query tree. DTD constraints for example can determine that certain configurations are illegal and can thus be eliminated from processing. In case of artificial ordering, the data set is initially pre-processed by rearranging the sibling nodes so that they respect the ordering. Nodes in the given twig query are also rearranged in a similar manner so that the matching can be done. For example, consider the query Q_{31} with 6 different child nodes. There are a total of $6! = 720$ different configurations to evaluate. When a label ordering, say a numerical ordering $8650 < 8680 < ... < 9400$, is placed on both the data set and the query then we need to evaluate only a single configuration. In fact, the label ordering gave a single valid configuration for every query in our workload.

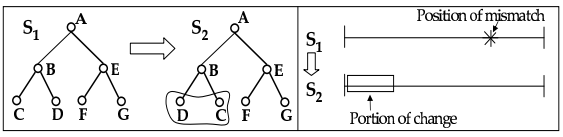

Figure 8: Unordered matching - A heuristic

When the application of constraints and orderings is not possible, one can employ some heuristics. For example, the naive approach can be improved by doing *a loop inversion* in the second step – for each tree in the database, process all configurations in C to find their matches. A tiling-like approach can then be used to exploit the fact that many configurations are quite similar. For example, Figure 8 shows two configurations S_1 and S_2. S_2 can be eliminated from processing because the position of mismatch in S_1 is away from the portion of the sequence that is changed in S_2. Such a heuristic complemented with a suitable ordering of configurations in C can greatly improve the performance. We are currently investigating these and other ways to extend LCS-TRIM to handle unordered matches.

4.4.2 NOT Predicates

Consider the twig query, NQ=//*supplier*/[NOT(//*store*-)]//*part* [31]. It selects the *part* elements with an ancestor *supplier* that has no descendant named *store*. There exist several ways to incorporate such constraints in query processing [18, 31] . A simple but inefficient approach is to divide the query into two sub queries – one with the tag (*store*) and one without the tag. Then take the difference between the results from those two sub queries.

LCS-TRIM handles NOT predicates by first constructing the R-matrix as described in Section 3.2.1, by ignoring the NOT-predicate on *store* tag. It checks for following two conditions to find a tree T that does not have the tag *store*. i) the length of LCS between T and NQ should be $m-1$. ii) the number of dominant matches in the row, in R, corresponding to *store* should be zero. Moreover, the structure agreement checks ignore the *store* tag in NQ and proceed to its parent nodes. More involved algorithms ought to be designed to support nested NOT predicates.

4.4.3 Approximate Query Matching

In case a query does not have any matches, instead of returning no results it would be nice to have a query engine that can return approximate results. Approximation can either be structural or content-oriented i.e., values. LCS-TRIM can be modified with a simple cost-based model to

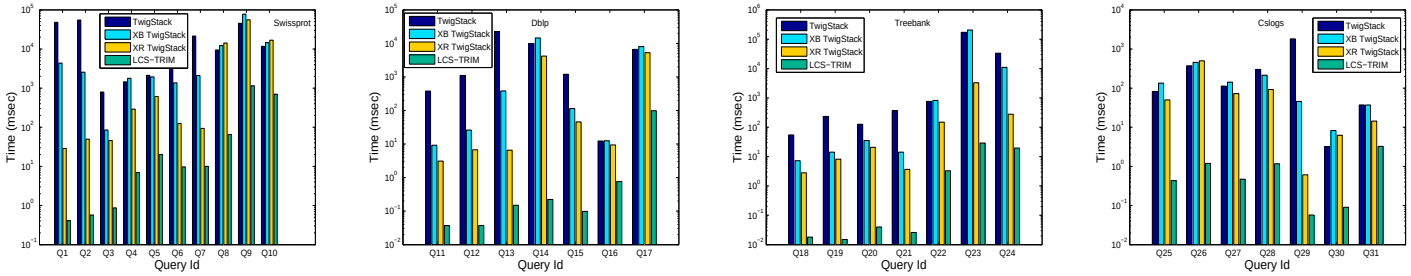

Figure 7: Performance comparison with TwigStack based algorithms

retrieve such approximate results. Each mismatch (either in structure or in content) is tagged with certain cost and the tree matching, both subsequence and structure matching, is continued as long as the total cost of mismatch is less than a pre-defined threshold. A domain expert can define the threshold and also the type and cost of mismatches.

5. CONCLUSIONS

In this paper, we have proposed a dynamic programming based method for indexing and querying a database of trees – with specific applications to XML data. Our approach relies on novel sequence based representations and a variant of the classical longest common subsequence matching algorithm to enumerate matches in the database. A series of optimizations are also designed to prune out false candidate matches early in the search process enabling efficient matching and retrieval. The simple array based data structures used by our approach are not only cache conscious but also make our algorithms amenable for parallel processing on emerging multi-core server architectures. The simple inverted tree index complements our efficient algorithms in achieving up to three orders of magnitude speedup over PRIX and TwigStack based approaches. We also present various extensions where the user can specify constraints, which can be pushed deep into the process to make the search more efficient and specify a mechanism wherein approximate matches may be returned by the system. We are currently examining how the out-of-core performance of our approach can be improved through strategies such as partitioning and tiling. We are also exploring new methods for unordered matching, approximate matching, and for incorporating NOT predicates.

Acknowledgments: We would like to thank Prof. Vassilis Tsotras and also the reviewers of this paper for their useful comments and suggestions in improving this paper.

6. REFERENCES

[1] S. Abiteboul et al. The Lorel query language for semistructured data. In *IJDL*, 1(1):68–88, 1997.
[2] S. Al-Khalifa et al. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In *ICDE*, pages 141–152, 2002.
[3] L. Bergroth et al. A Survey of Longest Common Subsequence Algorithms. In *SPIRE*, pages 39–40, 2000.
[4] N. Bruno et al. Holistic Twig Joins: Optimal XML Pattern Matching. In *ACM SIGMOD*, pages 310–321, 2002.
[5] D. Chamberlin et al. XQuery 1.0: An XML Query Language. *W3C Working Draft*, 7, 2001.
[6] D. Chamberlin et al. Quilt: An XML Query Language for Heterogeneous Data Sources. In *WebDB*, pages 1–25, 2000.
[7] S. Chien et al. Efficient Structural Joins on Indexed XML Documents. In *VLDB*, pages 263–274, 2002.
[8] C. Chung et al. APEX: An Adaptive Path Index for XML Data. In *SIGMOD*, pages 121–132, 2002.
[9] J. Clark et al. XML Path Language (XPath) Version 1.0. 1999.
[10] B. Cooper et al. A Fast Index for Semistructured Data. In *VLDB*, pages 341–350, 2001.
[11] T.H. Cormen et al. Introduction to Algorithms. *MIT Press, Cambridge, MA*, 2001.
[12] R. Goldman and J. Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *VLDB*, pages 436–445, 1997.
[13] T. Grust. Accelerating XPath Location Steps. In *SIGMOD*, pages 109–120, 2002.
[14] H. Prasad et al. Efficient Indexing and Querying of XML Data Using Modified Prüfer Sequences. In *CIKM*, pages 288–299, 2005.
[15] Daniel S. Hirschberg. Algorithms for the Longest Common Subsequence Problem. In *J. ACM*, 24(4):664–675, 1977.
[16] H. Jiang et al. XR-Tree: Indexing XML Data for Efficient Structural Joins. In *ICDE*, pages 253–264, 2003.
[17] H. Jiang et al. Holistic Twig Joins on Indexed XML Documents. In *VLDB*, pages 273–284, 2003.
[18] E. Jiao, T.W. Ling, and C.Y. Chan. PathStack¬: A Holistic Path Join Algorithm for Path Query with Not-predicates on XML Data. In *DASFAA*, 2005.
[19] R. Kaushik et al. Covering Indexes for Branching Path Queries. In *SIGMOD*, pages 133–144, 2002.
[20] H. Li et al. An Evaluation of XML Indexes for Structural Join. In *SIGMOD Record*, 33(3):28–33, 2004.
[21] Q. Li and B. Moon. Indexing and querying XML data for regular path expressions. In *VLDB*, pages 361–370, 2001.
[22] J. Lu et al. Efficient processing of ordered XML twig pattern. In *DEXA*, pages 300–309, 2005.
[23] M. Moro et al. Tree-pattern Queries on a Lightweight XML Processor. In *VLDB*, pages 205–216, 2005.
[24] J. Pei et al. Pushing Convertible Constraints in Frequent Itemset Mining. In *DMKD*, 8(3):227–252, 2004.
[25] S. Picciotto. How to Encode a Tree. *Ph.D. thesis, UCSD*, 1999.
[26] R. Praveen and M. Bongki. PRIX: Indexing and querying xml using prüfer sequences. In *ACM TODS*, 31(1):299-345, 2006.
[27] H. Prüfer. Neuer Beweis eines Satzes über Permutationen. *Archiv für Mathematik und Physik*, 27:742–744, 1918.
[28] R. Wagner and M. Fischer. The String-to-String Correction Problem. In *JACM*, pages 168–173, 1974.
[29] H. Wang et al. ViST: A Dynamic Index Method for Querying XML Data by Tree Structures. In *SIGMOD*, pages 110–121, 2003.
[30] H. Wang and X. Meng. On the Sequencing of Tree Structures for XML Indexing. In *ICDE*, pages 372–383, 2005.
[31] T. Yu et al. TwigStackList¬: A Holistic Twig Join Algorithm for Twig Query with Not-predicates on XML Data. *Springer LNCS*, 3882:249, 2006.
[32] M.J. Zaki. Efficiently Mining Frequent Trees in a Forest. In *SIGKDD*, pages 71–80, 2002.
[33] C. Zhang et al. On Supporting Containment Queries in Relational Database Management Systems. In *SIGMOD Record*, 30(2):425-436, 2001.
[34] N. Zhang et al. A Succinct Physical Storage Scheme for Efficient Evaluation of Path Queries in XML. In *ICDE*, pages 54–65, 2004.
[35] P. Zezula et al. Tree signatures for XML querying and navigation. In *Springer LNCS*, 2824:149-163, 2003.
[36] P. Zezula et al. Tree Signatures and Unordered XML Pattern Matching. In *SOFSEM*, pages 122–139, 2004.