Approximate XML Joins

Sudipto Guha [*]
University of Pennsylvania
sudipto@cis.upenn.edu

H. V. Jagadish[†]
University of Michigan
jag@eecs.umich.edu

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

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

Ting Yu [‡]
University of Illinois
tingyu@cs.uiuc.edu

ABSTRACT

XML is widely recognized as the data interchange standard for tomorrow, because of its ability to represent data from a wide variety of sources. Hence, XML is likely to be the format through which data from multiple sources is integrated.

In this paper we study the problem of integrating XML data sources through correlations realized as join operations. A challenging aspect of this operation is the XML document structure. Two documents might convey approximately or exactly the same information but may be quite different in structure. Consequently approximate match in structure, in addition to, content has to be folded in the join operation. We quantify approximate match in structure and content using well defined notions of distance. For structure, we propose computationally inexpensive lower and upper bounds for the tree edit distance metric between two trees. We then show how the tree edit distance, and other metrics that quantify distance between trees, can be incorporated in a join framework. We introduce the notion of reference sets to facilitate this operation. Intuitively, a reference set consists of data elements used to project the data space. We characterize what constitutes a good choice of a reference set and we propose sampling based algorithms to identify them. This gives rise to a variety of algorithmic approaches for the problem, which we formulate and analyze. We demonstrate the practical utility of our solutions using large collections of real and synthetic XML data sets.

1. INTRODUCTION

XML is widely recognized as the data interchange standard for tomorrow, in particular because of its ability to represent data from a wide variety of sources. Hence, XML is likely to be the language in which to integrate data from multiple sources. Data of string type are prevalent in XML, thus the traditional inconsistencies between string attributes, such as mis-spelling, will persist in the XML world as well. Correlating XML data sources however, has to cope with additional complexities due to the structure of XML documents, which cannot be ignored. Autonomous data sources may contain the same data, but may have differences in structure. It is important to be able to correlate such data. Even when data sources have the same Document Type Descriptor (DTD), they may not have identical tree structure due to the presence of optional elements and attributes.

[*]Part of this work performed while the author was with AT&T Labs.

[†]Work supported in part by NSF under grant IIS-0002356.

[‡]Work performed while the author was visiting AT&T Labs.

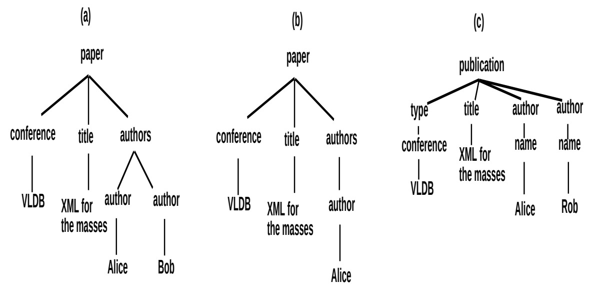


Figure 1: Example XML documents

EXAMPLE 1. *Figure 1 presents three XML documents. Documents (a) and (b) originate from the same DTD, but document (c) is an instance of a different DTD. It is apparent that all three documents describe the same publication. An application trying to integrate data from various sources will have to face several challenges in this example. In document (b) author* `Bob` *is not listed as one of the authors of the paper* "XML for the masses". *Based on the remaining elements and* **PCDATA** *fields however, documents (a) and (b) represent the same paper. Document (c) represents the same paper as documents (a) and (b) but originates from a different DTD. Besides different labels for elements (e.g.,* `publication` *versus* `paper`*), structural differences exist (e.g., element* `authors` *is not present, but element* `name` *is present) as well as spelling inconsistencies (e.g.,* **Rob** *versus* **Bob***).*

To address such difficulties, we need efficient techniques for *approximately matching* XML documents, based on the tree structured content of specified sub-elements of XML documents. Whenever one deals with notions of approximate matching, one has to specify a distance metric between the approximated entities that effectively quantifies the approximate match. Such a metric has to be amenable to efficient computation and general enough to encompass various types of differences between XML documents. As Example 1 demonstrates, there is a clear need for metrics that can handle inconsistencies, both in structure as well as content.

XML documents are ordered labeled trees; the problem of defining the distance between two ordered labeled trees has received much attention in combinatorial pattern matching and the notion of *tree edit distance* has been developed [16]. This distance is a a natural generalization of *edit distance* from the domain of strings. Informally, the tree edit distance between two trees is the minimum number of operations (node insert, delete, relabel) required to transform one tree to the other. A variety of computationally expensive algorithms have been proposed for computing tree edit distance between two trees. Such algorithms can serve as the basis of approximate tree match quantification. For example, in Figure 1 trees (a) and (b) are at tree edit distance 2 apart. Insertion of the subtree `author-Bob` in (b) will transform (b) to a document that matches (a) perfectly. Our goal, however, is to embed such approximate tree match algorithms in a broader correlation framework between XML data sources. Clearly, there is a requirement for efficient correlation (join) algorithms that can dispense with the computation of the exact tree edit distance between all possible input pairs, when possible.

In this paper, we initiate a formal study of join algorithms between XML data sources. We propose metrics and algorithms for this problem. There exists a large body of work dealing with the computation of string edit distance and its adaptation in a join framework [6, 7, 13]. We focus the bulk of this paper on the novel aspect of adapting tree edit distance computation (matching based on structural difference) into a join framework. Our algorithms are the first to address this problem and lay the foundations for this important area. More specifically, we make the following contributions:

- Computing tree edit distance between a pair of XML documents turns out to be a very expensive operation. Recognizing this problem, we present upper and lower bounds for the tree edit distance between XML document pairs. Such bounds are much more efficient to compute. Using these bounds as a basis, we develop efficient filtering techniques that often avoid the need to compute the tree edit distance between a pair of XML documents.

- We present algorithms for the efficient execution of approximate joins between XML documents using the tree edit distance as a join condition. Our algorithms are based on sampling and can effectively reduce the volume of data examined during the join operation. Our algorithmic framework is easily adaptable to *any metric* one wishes to apply for quantifying differences between XML documents.

- We present an experimental evaluation of our techniques using both real and synthetic data sets, analyzing the performance tradeoffs inherent in our approach.

This paper is organized as follows. Section 2 discusses work related to the problems addressed in this paper. In section 3 we provide definitions necessary for the bulk of the paper and formally define the problem considered in the paper. Section 4 presents lower and upper bounds for tree edit distance. In section 5 we introduce a transform, based on the notion of *reference sets*, that effectively reduces the problem of approximately joining XML documents to that of joining vectors in a metric space. Section 6 elaborates on the properties of the resulting metric space and proposes novel algorithms for the approximate join problem between XML data sources. In particular, we show how various bounds can be utilized in this framework and give rise to various algorithmic approaches. In section 7 we present a thorough experimental evaluation measuring and comparing the performance of the algorithms proposed herein for real and synthetic data sets. Finally, section 8 concludes the paper and outlines problems of interest for further study in this direction.

2. RELATED WORK

Matching strings approximately is a problem of central interest in pattern matching and a variety of algorithms exist to solve the problem in several ways [11]. The notion of *edit distance* is a fundamental measure used to quantify distance between strings [9]. Approximately matching collections of strings is a problem of central interest in the context of data integration and cleansing (see e.g., [6, 7, 13]). Join algorithms have been proposed to identify pairs of approximately matching strings [6, 7] and commonly reduce to traditional join algorithms deployed in relational systems.

A large body of work in combinatorial pattern matching deals with the problem of identifying the distance between ordered labeled trees [12, 1]. A very general notion of distance between trees can be expressed using the *tree edit distance* [1, 12]. Several algorithms have been developed to identify the tree edit distance between pairs of trees [16]. The problem of matching trees can be arbitrarily hard depending on the specific cost model adopted (see [16] and references therein for a comprehensive treatment of the subject).

With the popularity of the semistructured data model, several algorithms have been introduced to identify changes in pairs of semistructured or XML documents [3, 2, 10, 5]. Such algorithms focus on identifying changes in versions of a document and introduce cost models to quantify change, based on assumptions specific to the particular application. Since the version of the problem targeted in those works is very hard, several heuristics are proposed and evaluated. In this paper, we adopt the notion of tree edit distance due to its generality, and simplicity of operations that offer a conceptually appealing quantification of the distance between trees. We emphasize however that the algorithms presented herein can be adapted easily to *any* distance function as long as it is a *metric*. For example, previously proposed change detection techniques [10, 3] can be used as a basis to construct distance functions between trees. As long as these new notions of distance have metric properties, they can be easily incorporated in our framework.

3. PRELIMINARIES

Let Σ be an alphabet of size $|\Sigma|$. Let $\epsilon \notin \Sigma$ represent the null symbol. For a DTD D, let Δ be the set of all ordered labeled trees, well formed under D. Without loss of generality we will assume that all node literals and atomic values of D as well as element symbols are defined over Σ.

In the domain of strings, the notion of *edit distance* has been widely applied to quantify differences between strings.

DEFINITION 1 (EDIT DISTANCE). *The* edit distance *between two strings σ_1, σ_2, $ed(\sigma_1, \sigma_2)$ is defined as the minimum number of edit operations (i.e., insertions, deletions, substitutions) on single characters required to transform one string into another.*

Edit distance is a well studied distance metric with a plethora of applications in various fields of computer science. Given two strings σ_1 and σ_2, there exists a well known algorithm [9] to compute the edit distance between them in $O(|\sigma_1||\sigma_2|)$ time and space. Edit operations are assumed to be unit cost. Non-unit costs and more general edit operations are possible, and have been studied extensively in a significant stream of research that followed. Notice that $ed()$ is a metric.

A generalization of string edit distance is the notion of *tree edit distance* [1] defined for quantifying differences between pairs of ordered labeled trees. Let T_1, T_2 be two well formed XML documents, under the same or different DTD's. They can be conceptually represented (after parsing) as ordered labeled trees, where node labels include element tags, **PCDATA** values, attribute names and attribute values. The nesting of the elements is reflected in the tree structure. *Tree edit distance* is defined as follows:

DEFINITION 2 (TREE EDIT DISTANCE). *Given two trees T_1, T_2, the* tree edit distance *between them, TDIST(T_1, T_2), is defined as the minimum cost sequence of tree edit operations (i.e., node insertions, deletions and label substitutions) on single tree nodes, required to transform one tree to another.*

Computing the tree edit distance between trees is a well studied problem as well. Given a tree T, let $h(T)$ denote its height. For two trees T_1, T_2, there exists a well known algorithm to compute the tree edit distance between them, running in $O(|T_1||T_2|h(T_1)h(T_2))$ time and $O(|T_1||T_2|)$ space [16]. Once again, notice that TDIST is a metric as well. As in the case of string edit distance, numerous variants of tree edit distance have been introduced. Such variants can be incorporated in our algorithms as long as the distance they compute is a metric.

We briefly describe algorithm $treedist()$ below that computes the TDIST between two trees, and refer the reader to [16] for a comprehensive treatment. Typically, the cost of each edit operation is assumed to be one unit, and the cost of a sequence of edit operations is simply the sum of its component operation costs.

Notation. Let T be an ordered labeled tree. Order is obtained by a left to right postorder numbering of the nodes in T [1]. An ordered sub forest of T is a collection of subtrees of T appearing in the same order as they appear in T. Let $t[i]$ represent the i node of T, n_i the children of i, $T[i]$ the subtree rooted at $t[i]$ and $F[i]$ the sub forest obtained by deleting from $T[i]$ node $t[i]$. We represent an edit operation as $\gamma(a \rightarrow b)$. The edit operation is an *insert* operation if $a = \epsilon$, a *delete* operation if $b = \epsilon$ and a substitution operation if $a \neq \epsilon$ and $b \neq \epsilon$. Notice that this makes $\gamma()$ a metric. These operations are assumed to have unit costs.

Tree Edit Distance Computation. Let $treedist(T_1, T_2)$ denote the algorithm that computes the tree edit distance TDIST between trees T_1, T_2. The algorithm constructs a mapping M between the nodes of the two trees; this mapping dictates correspondences between nodes of the two trees. The goal is to compute the mapping with the minimum cost subject to the specific cost model. The mapping consists of pairs of integers (i, j) such that:

- $1 \leq i \leq |T_1|$ and $1 \leq j \leq |T_2|$
- For any pairs $(i_1, j_1), (i_2, j_2) \in M$
    1. $i_1 = i_2$ if and only if (iff) $j_1 = j_2$
    2. $t_1[i_1]$ is to the left of $t_1[i_2]$ iff $t_2[j_1]$ is to the left of $t_2[j_2]$ (sibling order preserving)
    3. $t_1[i_1]$ is an ancestor of $t_1[i_2]$ iff $t_2[j_1]$ is an ancestor of $t_2[j_2]$ (ancestor order preserving)

Figure 2 illustrates this algorithm with an example. The matching M generated in this case is depicted with lines in the figure. Nodes that are not matched have to be considered for insertions or deletions. Nodes that are matched have to be considered for relabeling. $treedist()$ evaluated between the two trees in the figure returns a value of 3 (delete node B, insert node H, relabel C to I).

[1]Without loss of generality, we use postorder numbering. Preorder would work as well.

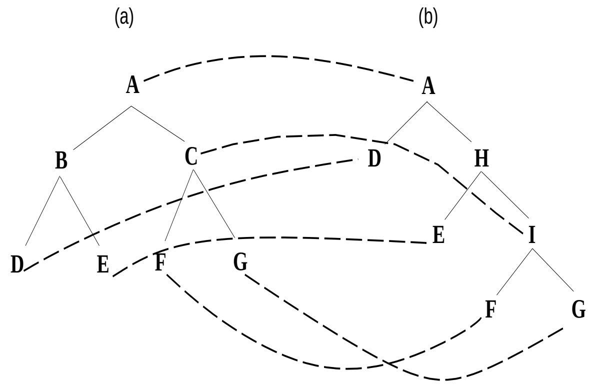


Figure 2: Example $treedist()$ mapping between two trees

3.1 Problem Definition

Let S_1 and S_2 be two sources of XML data originating from the same or different DTD's. We are seeking algorithms to perform an approximate join operation between the sources using tree edit distance (TDIST) as a join predicate applied on pairs of subtrees of the XML documents, corresponding to the sub elements that need to be matched in the XML documents. For simplicity of exposition, we assume in this paper that entire documents need to be matched. More formally:

DEFINITION 3 (APPROXIMATE TDIST JOIN). *Given two XML data sources, S_1 and S_2 and a distance threshold τ, let TDIST(d_1, d_2) be a function that assesses the tree edit distance between two documents $d_1 \in S_1$ and $d_2 \in S_2$. The approximate join operation between two sources of XML documents reports in the output all pairs of documents $(d_1, d_2) \in S_1 \times S_2$ such that TDIST$(d_1, d_2) \leq \tau$.*

Variants of this basic problem are also possible; for example we might require the distance between two XML documents to be in a range of user specified distance thresholds. Our algorithms can provide solutions to these variants of the problem as well. We will suppress this discussion for brevity. TDIST as defined assumes edit operations of unit cost. Incorporating the capability to assess string edit distance between string node labels into TDIST, can be performed by evaluating the edit distance $ed()$, between two node labels of string type. The edit function $\gamma()$ can be modified for this purpose to assess $ed()$ between two nodes, whenever these nodes have string labels (e.g., CDATA,PCDATA etc). Since $ed()$ is a metric, the metric properties of function $\gamma()$ are preserved (a property required by our proposal). Handling edit operations on strings is an additional function call in the tree edit distance computation $treedist()$ whenever the nodes have string labels. This way of incorporating string edit distance in our framework does not affect our algorithms or their correctness. In a similar fashion, an ontology hierarchy can be incorporated, matching/comparing tag names. In order to simplify the presentation in the remainder of this paper, edit operations are assumed to be unit cost.

Executing an approximate TDIST join operation efficiently faces two main challenges. First, evaluation of the TDIST function between two documents is a very expensive operation; in the worst case it is an $O(n^4)$ operation for trees of size $O(n)$. Clearly, even for small values of n, straightforward evaluation of the function is computationally prohibitive. Second, traditional wisdom in join algorithms (sort merge, hash joins etc) does not extend easily in this application domain. Novel join processing techniques are required to deal with the intrinsic complexities of approximate matching of

XML documents.

Overview of our approach. We will present our proposal in the following steps:

- We will develop lower and upper bounds for the TDIST function (Section 4). These bounds are computationally less expensive than the evaluation of TDIST and they serve as a basis for significant reduction in the overall computation costs. We can utilize these bounds to design inexpensive filters for TDIST. Application of our bounds provides a fast way to decide if the application of TDIST to a pair of documents of interest is within our distance threshold. For example, the pair can be dismissed if the bounds indicate that it is not, saving the expensive TDIST evaluation on the pair.
- We will present (Section 5) algorithms for evaluating joins between pairs of XML data sources utilizing our bounds for TDIST. Although bounding TDIST can reduce the computation for a single pair, one has still to improve on the overhead of examining all pairs of documents. We develop sampling based algorithms for this problem. Our algorithms guarantee no false dismissals and effectively transform the problem of joining XML documents to that of performing a join operation in a numeric vector space. Using this transform, along with our bounds and properties of the resulting vector space, we present a family of algorithms for the problem of approximately joining XML data sources (Section 6). We subsequently evaluate our proposed algorithms tuning various parameters of interest and observe their comparative performance (Section 7).

All the proofs of theorems and lemmas are omitted due to space limitations.

4. BOUNDING TREE EDIT DISTANCE

Computation of the tree edit distance between two trees T_1, T_2, requires time $O(|T_1||T_2|h(T_1)h(T_2))$ (where $|T_i|$ is the number of nodes in tree T_i and $h(T_i)$ is the the height of T_i). Given large, deeply nested XML document trees, such a computational cost may be too high. In this section we devise computationally efficient techniques to obtain both lower and upper bounds for TDIST. In Section 6, we will show how to use these bounds to solve our approximate join problem faster.

4.1 Deriving Lower Bounds

Let T be an ordered labeled tree. Let $pre(T)$ denote the preorder traversal of T and $post(T)$ its postorder traversal. Both $pre(T)$ and $post(T)$ can be viewed as strings over Σ^*. For two trees T_1, T_2,

LEMMA 1. *If $pre(T_1) \neq pre(T_2)$ or $post(T_1) \neq post(T_2)$, then $T_1 \neq T_2$.*

Intuitively the lemma states that if there is a difference between the prefix or postfix traversals of the trees, then there has to be a difference in the trees as well. There are natural examples showing that there are differences in the trees that are not reflected in the tree traversals. This is acceptable since we are searching for a lower bound.

LEMMA 2. *If the trees are at edit distance k, then the maximum edit distance between their preorder or postorder traversals is at most k.*

Based on lemmas 1,2 we obtain the following relationship between TDIST and the string edit distances corresponding to the two traversals:

THEOREM 1. *Let T_1, T_2 be ordered labeled trees. Then*

$$max(ed(pre(T_1), pre(T_2)), ed(post(T_1), post(T_2)))$$
$$\leq TDIST(T_1, T_2)$$

We illustrate this bound through the following example.

EXAMPLE 2. *Consider again the trees (a),(b) of Figure 2. We have $pre((a)) = ABDECFG$ and $post((a)) = DEBFGCA$. Similarly $pre((b)) = ADHEIFG$ and $post((b)) = DEFGIHA$. Clearly,*

$$max(ed(pre((a)), pre((b))), ed(post((a)), post((b))))$$
$$= max(3, 3) = 3 \leq treedist((a), (b))$$

We will denote the lower bound of theorem 1 as LBDIST(T_1, T_2) for trees T_1, T_2. Notice that for strings (preorder, postorder representations) of length $O(n)$, $ed()$ can be computed in $O(n^2)$ worst case time and space. This can be substantially better than computing the tree edit distance directly.

4.2 Deriving Upper Bounds

We construct an upper bound UBDIST of TDIST by restricting the freedom of choices algorithm $treedist$ has when computing the optimal set of operations to match two trees. More precisely, we will impose additional constraints on the kinds of relationships the algorithm maintains when devising the optimal set of operations. This effectively reduces the search space and enables the development of a faster algorithm. We note, that such a measure was introduced by Zhang [15], where an algorithm related to the one presented herein was proposed.

Recall that the minimum cost mapping M that algorithm $treedist$ obtains is sibling and ancestor order preserving. For the upper bound construction, we require that the mapping M also preserves ancestor order for the *lowest common ancestor of pairs of nodes*. For any triple $(t_1[i_1], t_2[j_1]), (t_1[i_2], t_2[j_2]), (t_1[i_3], t_2[j_3]) \in M$, let $lca()$ be the lowest common ancestor function. We require that mapping M respects:

CONDITION 1. *$t_1[lca(t_1[i_1], t_1[i_2])]$ is a proper ancestor of $t_1[i_3]$ iff $t_2[lca(t_2[j_1], t_2[j_2])]$ is a proper ancestor of $t_2[j_3]$.*

Intuitively, the new requirement ensures that two distinct subtrees of T_1 will be mapped to two distinct subtrees of T_2. These requirements are satisfied by a dynamic programming algorithm related to the one computing edit distance between two trees [16]. Algorithm $DistinctTreeEditDistance$ computing UBDIST between a pair of trees, is presented in Figure 3. It constructs a minimum cost mapping between the nodes of the two trees, respecting the constraints of condition 1, as well as the constraints satisfied by algorithm $treedist$. For any pair of subtrees it accounts for the various types of correspondence between the tree node pairs. The first two equations in formula (3) account for the case that exactly one of the nodes under consideration does not belong in the mapping; in this case it considers the optimum cost to match a descendent subtree of that node. The last equation accounts for the cost of matching two subtrees (matching the respective forests after matching the root nodes). Formula (2) in Figure 3 is similar to formula (3) but for the case of forests in the two trees. The last equation in formula (2) considers the cost of matching the children nodes of a pair under consideration. Formula (1), in Figure 3, keeps track of this cost.

Consider the example depicted in Figure 4. The match generated by algorithm $treedist$ is shown in dashed lines. The TDIST computed is 3. The UBDIST computed with algorithm

```
Algorithm $DistinctTreeEditDistance(T_1, T_2)$ {
$D[|T_1|][|T_2|]$ records distances of subtrees (substructures) of $T_1, T_2$
$ED[|T_1|][|T_2|]$ records distances between children of a pair of nodes
$1 \leq i \leq |T_1|, 1 \leq j \leq |T_2|$
Initialize $D[i][\epsilon], D[\epsilon][j] \forall i, j$
    for $i$ = 1 to $|T_1|$ {
     for $j$ = 1 to $|T_2|$ {
      for $k$ = 1 to $n_i$ { /* the children of i */
       for $l$ = 1 to $n_j$ { /* the children of j */
        $(1) ED[k][l] = min \begin{cases} ED[k][l-1] + D[\epsilon][T_2[j_l]] \\ ED[k-1][l] + D[T_1[i_k]][\epsilon] \\ ED[k-1][l-1] + D[T_1[i_k]][T_2[j_l]] \end{cases}$
        $(2) D[F_1[i]][F_2[j]] = min \begin{cases} D[\epsilon][F_2[j]] + min_{1 \leq l \leq n_j}(D[F_1[i]][F_2[j_l]] - D[\epsilon][F_2[j_l]]) \\ D[F_1[i]][\epsilon] + min_{1 \leq k \leq n_i}(D[F_1[i_k]][F_2[j]] - D[F_1[i_k]][\epsilon]) \\ ED[n_i][n_j] \end{cases}$
        $(3) D[T_1[i]][T_2[j]] = min \begin{cases} D[\epsilon][T_2[j]] + min_{1 \leq l \leq n_j}(D[T_1[i]][T_2[j_l]] - D[\epsilon][T_2[j_l]]) \\ D[T_1[i]][\epsilon] + min_{1 \leq k \leq n_i}(D[T_1[i_k]][T_2[j]] - D[T_1[i_k]][\epsilon]) \\ D[F_1[i]][F_2[j]] + \gamma(t_1[i] \rightarrow t_2[j]) \end{cases}$
```

Figure 3: Algorithm $DistinctTreeEditDistance$ computing UBDIST

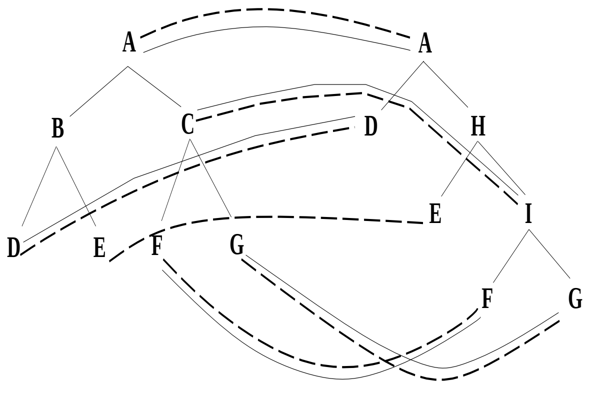


Figure 4: Example matching between two trees

$DistinctTreeEditDistance$ is 5 (delete B, delete E, insert H, insert E, relabel C to I), however. The match generated by the algorithm is shown in solid lines. Unlike algorithm $treedist$ the algorithm will not establish the match between E's since this would violate condition 1. We claim the following:

THEOREM 2. *There exists an $O(n^2)$ algorithm to compute the UBDIST distance between a pair of trees with $O(n)$ nodes each.*

Proof:
It is evident that $\sum_{i=1}^{|T_1|} \sum_{j=1}^{|T_2|} O(n_i \times n_j) \leq O(|T_1| \times |T_2|)$ □

It is easy to see that UBDIST is reflexive and obeys the triangle inequality, thus it is a metric. We establish the upper bound relationship between TDIST and UBDIST with the following theorem:

THEOREM 3.

$$For\ a\ pair\ of\ trees\ T_1, T_2, \textit{TDIST}(T_1, T_2) \leq \textit{UBDIST}(T_1, T_2)$$

In this section, we have shown how to compute fast lower and upper bounds on the distance between any pair of trees. These metrics, LBDIST and UBDIST respectively, will be used in the efficient algorithms we derive in Section 6.

5. REDUCTION INTO A METRIC SPACE USING REFERENCE SETS

In the preceding section we focused on the time taken to compute the distance between two trees, and presented efficient techniques to bound this distance. In this section, we will devise a technique to minimize the number of pairwise distance computations required to evaluate the approximate tree join of two sets of XML trees.

Let S_1, S_2 be two sets of XML document trees, say from different data sources, between which we wish to compute a join. Let $K \subset S_1 \cup S_2$ be a chosen set of XML documents, which we will refer to as a *reference set*. We will defer discussion on how best to determine K until Section 5.1.

Let $d_i \in S_1, d_j \in S_2$ be two XML documents. We will construct a vector for each document consisting of the distances of each document to the XML documents in K. This could be any arbitrary metric distance function, in general, and not just the TDIST function. To emphasize this point, we use $dist$ rather than TDIST in this section, even though our interest is in the special case when the distance of each document to the documents in K are quantified using TDIST.

Let $k_1, \ldots k_{|K|}$ be an arbitrary ordering of the reference set K. Let v_i (v_j) be the vector for d_i (d_j). Clearly each vector is of dimensionality $|K|$. Further let $v_{i\ell} = dist(d_i, k_\ell), 1 \leq \ell \leq |K|$; similarly $v_{j\ell} = dist(d_j, k_\ell), 1 \leq \ell \leq |K|$. Since $dist()$ is a metric, the following is true by application of the triangle inequality:

$$\forall 1 \leq \ell \leq |K| \; |v_{i\ell} - v_{j\ell}| \leq dist(d_i, d_j) \leq v_{i\ell} + v_{j\ell} \quad (1)$$

Essentially the above procedure "projects" documents d_i, d_j onto the reference set. Provided that $dist()$ is a metric, the reference set acts as a set of basis elements defining a metric space. Assume that we wish to identify if the documents d_i,d_j are within a specific distance threshold, say τ, of each other, that is if $dist(d_i, d_j) \leq \tau$. The metric properties of $dist()$, namely equation 1 provides a way to reason about the pair's distance. It is evident due to equation 1 that if:

$$u_t = min_{\ell, 1 \leq \ell \leq |K|} v_{i\ell} + v_{j\ell} \leq \tau \quad (2)$$

then the pair is certainly within distance of τ. Similarly, if:

$$l_t = max_{\ell, 1 \leq \ell \leq |K|} |v_{i\ell} - v_{j\ell}| > \tau \quad (3)$$

then the pair cannot be within distance τ. Thus, the properties of the metric space, constructed using the reference set, provide us with additional capability to reason about the distance between a pair of XML documents. Depending on the relationship of l_t, u_t to τ one can conclude that the pair is below or above the desired threshold with certainty.

Clearly a projection of the entire XML sources S_1, S_2 can be obtained in a similar way. For a reference set K, let $S_1^{'}$ ($S_2^{'}$) denote the set of vectors v_i, $1 \leq i \leq |S_1|$ (v_j, $1 \leq j \leq |S_2|$) obtained after projecting each document $d_i \in S_1$ ($d_j \in S_2$) onto the reference set. For a metric $dist()$, provided that the reference set K is in memory, $S_1^{'}, S_2^{'}$ can be obtained with a single pass over the underlying XML data sources. Moreover, the projection achieved using the reference set effectively is a "dimensionality" reduction technique. Assume that the average document size in S is μ; the original data volume is $|S| \times \mu$ in this case. After projection, using a reference set of size $|K|$, the resulting data volume is $|S| \times (|K|+1)$ (we add one to account for the document id recorded in conjunction with each vector). If $|K|$ is small compared to μ, the reduction in data volume can be substantial, possibly enabling the vector sets to be memory resident.

5.1 Choosing the Reference Set

Given two sets of XML trees S_1, S_2 and a metric $dist()$ that assesses the distance between elements of the two sets, we seek to identify a reference set K. We would like this set to be as small as possible, since the amount of work we do is proportional to the size of this set; yet we would like the set to generate filters l_t, u_t that are as decisive as possible.

Given a distance threshold τ, assume $S = S_1 \cup S_2$ is divided into k clusters such that documents within a cluster have small distance (say less than $\frac{\tau}{2}$) and documents in different clusters have large distance (say larger than $\frac{3\tau}{2}$). In this case we will say that S is *well separated*. Suppose we choose one document d from a cluster containing n_1 documents in our reference set. Then any pair of documents in this cluster is within distance half the threshold from d. We would assert by the triangle inequality that the pair of documents constitute a valid pair (distance $\leq \tau$) and should be in the output, saving an evaluation of $dist()$ for this pair. Thus, we will save $n_1(n_1 - 1)/2$ such evaluations in total. Suppose that we now have a pair in which one document belongs to this cluster and the second document is in some other cluster. Then the distance of the point in the cluster and d is at most $\frac{\tau}{2}$; the distance of the point outside the cluster to d is at least $\frac{3\tau}{2}$. Again from the triangle inequality we can conclude that this pair is not close (distance $> \tau$), and avoid evaluating $dist()$. This will save us $n_1(|S| - n_1)$ comparisons. We have to distinguish the other pairs by the other points belonging to the reference set. Thus:

LEMMA 3. *If S is well separated the optimal strategy for constructing a reference set is to always choose a single point from each of the k largest clusters.*

Even if S is well separated into k clusters, however, we don't know apriori which are the k clusters that we should choose points from. It is evident, however, that every time we choose a sample from one of the clusters, we have to perform additional $|S|$ comparisons when constructing the vectors, but we also save a number of comparisons due to pruning via the triangle inequality. The optimal strategy minimizes the overall number of $dist()$ computations we perform. Let the optimal number of comparisons identified by the optimal strategy be W; We will show how to identify k clusters by sampling. Our strategy consists of first choosing a sample and then clustering the sample into k clusters with a clustering algorithm of choice [8, 17]. The following theorem shows how to identify k clusters in a well separated data set via sampling, such that the optimal number of comparisons we perform if we use a single point from each of these clusters identified as our reference set, is $(1+\epsilon)W$, for $\epsilon > 0$. We claim the following:

LEMMA 4. *If S is well separated into k clusters, a sample size of*

$$12 \frac{\sqrt{k|S|}}{\epsilon} \log |S|$$

is enough to identify all k such clusters with high probability.

In practice, however, although clustering might exist in the underlying data set, we don't expect it to be well separated. Moreover, clustering algorithms require the number of clusters as a parameter in the input and strive to derive the best clustering optimizing some measure function, with respect to the number of clusters specified. In our problem however, the user only specifies a distance threshold desired for the approximate join operation.

We build on the intuition gained by the preceding analysis and we propose the following strategy to identify the reference set. If the number of points k we choose in our reference set is known, we derive a sample size as computed in lemma 4. Clearly more samples are required since the data set is expected not to be well separated. We perform our clustering by repeatedly picking a point from the sample and covering all points within $\frac{1}{2}$ the user specified distance threshold. We then choose the k clusters which have the largest number of sample points and pick a random point from each to be in our reference set.

If we do not know the size of the reference set, we draw a sample according to lemma 4, of size at least $O(\sqrt{|S|} \log |S|)$ and we cluster to half the threshold as before. We then compute f_i as the fraction of points in the first i clusters. As i increases and f_i increases, we will be comparing $(1-f_i)^2 n^2$ pairs. Thus the number of comparisons decreases by ratio $(1-f_{i+1})^2/(1-f_i)^2$ and the size of the reference set increases by $1 + 1/i$ in size. We balance these two and choose $k \geq i \geq 2$ such that

$$\frac{(1-f_{i+1})^2}{(1-f_i)^2} > \frac{i}{i+1}$$

We will evaluate this strategy in section 7.

6. APPROXIMATE JOIN ALGORITHMS: BOUNDS + REFERENCE SETS

We present several algorithmic approaches for the efficient solution of the approximate join problem between XML data sources. We begin with a naive "current-practice" algorithm and enhance it in multiple ways by exploiting the techniques developed in the preceding sections.

6.1 A Baseline Algorithm

In relational databases, a well studied set of algorithms for computing joins between single valued attributes exists, like hash joins, sort merge joins and nested loop joins. Unfortunately, hash joins and sort merge joins do not carry over easily to the approximate XML join problem, as the join predicate makes use of a distance

```
Algorithm Bounds
  for each $d_1 \in S_1$ {
    for each $d_2 \in S_2$ {
      if (UBDIST$(d_1, d_2) \leq \tau$)
          output($(d_1, d_2)$)
      if (LBDIST$(d_1, d_2) \leq \tau$)
          if ($treedist(d_1, d_2) \leq \tau$)
            output($(d_1, d_2)$)
    }
  }
```

Figure 5: Algorithm `Bounds`

```
Approximate Join Algorithm
Construct $S_1^{'}, S_2^{'}$ using $dist()$ between
elements of $S_1, S_2$ and the reference set
vectors $v_i$, corresponding to documents $d_i$
    for each $v_i \in S_1^{'}$ {
      for each $v_j \in S_2^{'}$ {
        if ($UpperBound \leq \tau$)
        output($d_i, d_j$)
        if ($LowerBound \leq \tau$)
          if (TDIST$(d_i, d_j) \leq \tau$)
            output($d_i, d_j$)
      }
    }
```

Figure 6: Approximate Join Template

threshold which requires an evaluation of an expensive function between every input pair. This feature of the problem deems all algorithms that treat each joining attribute in isolation in their first step (e.g., hashing or sorting single attributes) inapplicable. A nested loops approach is readily applicable as it examines all pairs of input documents.

Distance based join algorithms, proposed in the field of spatial and multimedia databases, are also not easy to adapt in a straightforward way, as the underlying data type is a vector obtained through the application of various transforms. There is no obvious way of converting a document in isolation to a numeric vector.

Given two XML data sources S_1, S_2, a naive solution to the approximate join problem would compute a nested loop join between the two sources evaluating algorithm $treedist(d_i, d_j) \forall d_i \in S_1, d_j \in S_2$. Such an approach, has $O(|S_1| \times |S_2|)$ worst case I/O complexity. Provided that each document is of size $O(n)$ it invokes $treedist$ for each pair requiring $O(n^4)$ time in the worst case. Clearly such an approach has very high I/O and processor complexity but can serve as a baseline solution to this problem. We refer to this approach as `Naive (N)` in the sequel.

6.2 Improving the Baseline Using Bounds

A first improvement to the naive algorithm can be obtained by utilizing the upper and lower bounds introduced for TDIST. Let τ be the distance threshold specified for the join operation. From theorem 1, LBDIST$(d_1, d_2) \leq$ TDIST(d_1, d_2). A viable approach is to first evaluate LBDIST between the pair. If LBDIST$(d_1, d_2) > \tau$, we can safely prune this pair away, saving the invocation of the expensive $treedist(.)$ function. In case LBDIST$(d_1, d_2) \leq \tau$, however, the upper bound of theorem 3 could provide a quick way to decide if indeed the pair belongs to the answer set. If UBDIST$(d_1, d_2) \leq \tau$ then (d_1, d_2) belongs to the final answer and evaluating $treedist(d_1, d_2)$ is not required. In these cases, the lower and upper bounds can safely determine if a pair is not in the answer set (assuring no false negatives) or if a pair is in fact in the answer set (assuring no false positives). However, if LBDIST$(d_1, d_2) \leq \tau <$ UBDIST(d_1, d_2) the pair cannot be pruned or definitely included. In this case, one has to test if the pair is within distance τ by executing algorithm $treedist(.)$ on the pair. We refer to this improvement of algorithm `Naive` as `Bounds (B)` to signify the use of bounds. Pseudo code for this algorithm is given in Figure 5. Algorithm `Bounds` has the potential to reduce the processor time as evaluating the bounds is substantially cheaper than evaluating $treedist(.)$. Moreover, depending on how effective pruning is, the number of $treedist(.)$ computations can be substantially reduced. We will experimentally evaluate the effectiveness of these filters in Section 7.

6.3 Pruning Work with a Reference Set

Given a reference set K, first use TDIST to obtain $S_1^{'}, S_2^{'}$. Joining the two data sets takes place by application of Equation 1 on each pair. If the lower bound $l_t > \tau$, the pair can be pruned away. Otherwise, if the upper bound $l_u \leq \tau$, the pair belongs to the output. In the case $l_t \leq \tau < u_t$, application of $treedist()$ on the pair of corresponding XML documents is required to identify their true distance and decide if the pair belongs to the output. In this case, the actual documents are retrieved from secondary storage, based on their document identifiers and $treedist(.)$ is used to evaluate the actual distance. The advantage of this approach is that the bounds computed with Equation 1 for the TDIST between each pair, are exact since TDIST is used to compute the distances to the reference set. However, the algorithm has to perform $(|S_1| + |S_2|)|K|$ invocations of TDIST to compute the vector collections $S_1^{'}, S_2^{'}$.

We refer to this algorithm as `ReferenceSets (RS)` to signify that it uses a reference set to project the XML data sources, evaluating $treedist()$ between the elements of the sources and the elements of the reference set; it then utilizes the triangle inequality to prune the result space. The approach can be instantiated in Figure 6 using $dist =$ TDIST, $LowerBound = l_t$ and $UpperBound = u_t$ from Equation 1.

6.4 Applying Both Optimizations in Sequence

One can complement the above `RS` algorithm by the application of the lower and upper bounds introduced in theorems 1,3. If the bounds obtained by Equation 1 indicate that $treedist(.)$ has to be invoked between the pair under consideration to assess the exact distance, one can possibly avoid such execution by applying the computationally cheaper lower (LBDIST) and upper (UBDIST) bounds on the pair. Only if these bounds indicate that the pair could be in the result set, one should execute $treedist(.)$ on the pair to reach a conclusion.

We refer to this algorithm as `RSBounds (RSB)` to indicate the use of TDIST in the construction of the vector sets and the use of the l_t, u_t as well as the bounds of lemmas 1,3. It can similarly be instantiated in Figure 6 using $dist =$ TDIST and applying both l_t and LBDIST for LowerBound and both u_t and UBDIST for UpperBound.

6.5 Estimating Distances to the Reference Set

To lower the computational expense of evaluating $treedist(.)$ between the XML sources and the reference set, one could instead estimate the distance between them using LBDIST and UBDIST. This has the potential of reducing the computational expenses of this construction since we are evaluating much cheaper functions between the sources and the reference set. However, since these are bounds, once we use these in a manner that prevents false dismissals, we are left with potentially more pairs of elements between which we have to evaluate $treedist()$.

During construction of the vector sets, we construct two vectors for each document d_i. Vector v_i^l is a vector populated using LBDIST between the document and elements of the reference set; vector v_i^u is populated using UBDIST between the document and elements of the reference set. When computing the l_t, u_t bounds, equation 1 has to be modified to assure correctness in this case (no false positives or negatives). In particular:

$$\forall \ell 1 \leq \ell \leq |K| \;\; |v_{i\ell}^l - v_{j\ell}^u| \leq dist(d_i, d_j) \leq v_{i\ell}^u + v_{j\ell}^u \qquad (4)$$

Then u_t becomes $min_{\ell,1\leq\ell\leq|K|} v_{i\ell}^u + v_{j\ell}^u$ and l_t becomes $max_{\ell,1\leq\ell\leq|K|}|v_{i\ell}^l - v_{j\ell}^u|$. We refer to this combined algorithm as `RSCombined (RSC)` to signify the use of LBDIST and UBDIST and the application of the (modified) l_t, u_t bounds as well as the bounds of theorems 1,3. Notice that this algorithm doubles the size of the vector sets, since two vectors are constructed for each document. This algorithm can be similarly instantiated with the template of Figure 6.

7. EXPERIMENTAL EVALUATION

We implemented all the approaches proposed in this paper. In this section we present a comparative study varying parameters of interest. There are various parameters affecting the performance of our algorithms, and we conducted a comprehensive set of experiments to understand the impact of individual parameters to performance.

We used both synthetic and real data sets in our experiments. We choose to use synthetic data sets because it is easier to control the parameters and vary them on demand to isolate performance implications. We demonstrate the strength of our algorithms however, reporting performance results on real data sets. More specifically we used the following data sets in our study.

- **Data set A:** Synthetic data set constructed with the IBM XML data generator available through AlphaWorks. It consists of 500 randomly generated documents.

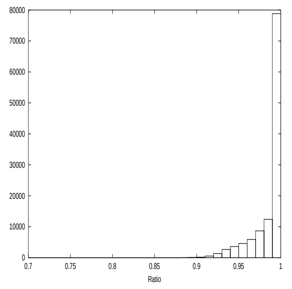
- **Data set B:** Synthetic data set constructed by merging documents from different runs of the IBM XML data generator. This data set was artificially constructed to contain 8 clusters of documents. It consists of 500 documents.

- **Real DBLP data:** We report experimental results on the entire conference collection of the DBLP database of size 55MB.

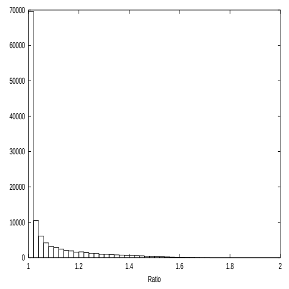
We first present accuracy and performance results for the LBDIST and UBDIST bounds to TDIST introduced in Section 4. Then we will present an evaluation of the algorithms presented in Section 6, varying parameters of interest. Finally, we will evaluate our proposal for the choice of a reference set and present results of the performance of our algorithms on real XML data sets highlighting the performance benefits of our proposal.

7.1 Evaluating Bounds

The results of the first experiment we present evaluates the quality of the LBDIST and UBDIST bounds to TDIST presented in Section 4. Figures 7(a)(b) present the results of the following experiment. For dataset A, we computed the pairwise distances using algorithm $treedist$ as well as the bounds to TDIST using LBDIST and UBDIST. We compute the ratio of the bound to TDIST and we construct a histogram of the number of pairs of documents falling



(a) Tightness of LBDIST bound



(b) Tightness of UBDIST bound

Figure 7: Evaluating the tightness of upper and lower bounds

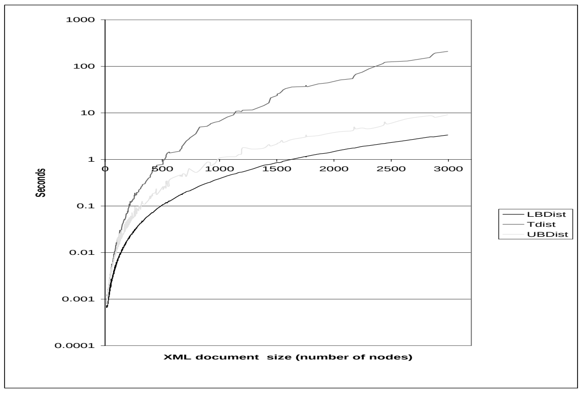


Figure 8: Time to compute $treedist$ and bounds as the document size increases

within each ratio range. Figure 7(a) shows a histogram of the number of document pairs versus the ratio of LBDIST to TDIST. Similarly, Figure 7(b) shows a histogram of the ratio of UBDIST to TDIST for all pairs. We can observe that in Figure 7(a) all pairs are above 0.9 with the bulk above 0.98. In Figure 7(b) all pairs are below 1.5 with the bulk of pairs below 1.2. Similar results were obtained for the other data sets. From the results of Figure 7 we can observe that both bounds are reasonably close to the actual distance obtained by algorithm $treedist(.)$. We next present the time taken to compute these bounds for various document sizes. Figure 8 presents the time (in logarithmic scale) required by all algorithms to compute the distance as the size of the XML document, in terms of number of nodes, increases. It is evident that as the size increases the bounds computations become progressively faster relative to the full computation, in accordance with our analytical expectations. (The Y-axis is on a logarithmic scale, and the curves are diverging, with a difference already of an order of magnitude for documents that are 500 elements).

7.2 Evaluating the Proposed Algorithms

There are two main parameters affecting the performance of the algorithms proposed. In the case of algorithms based on reference sets, the size and choice of the set is very important. The size of the set affects performance while building the vector collections and the choice of the reference set mainly determines the effect on pruning achieved. The distance threshold specified during the join operation affects the performance of baseline approaches as well as the performance of algorithms using reference sets. We vary both parameters in the sequel and we observe their performance implications.

In our first experiment, we investigate the impact of the reference set size to the performance of the algorithms making use of reference sets. Since we wish to vary the size of the reference set arbitrarily, we select it using uniform random sampling in this experiment and observe the trends. Figures 9(a)(b)(c) present the performance, as a fraction of the performance of Naive (N) of the three algorithms (ReferenceSets (RS), RSBounds (RSB), RSCombined (RSC)) as a function of the size of the reference set for data set A, for various distance thresholds. Algorithm Bounds (B) is not evaluated in this experiment as it does not make use of reference sets. Figures 9(d)(e)(f) present the results of the same experiment for data set B. Figure 9(a) presents the performance of RS. Two observations are evident. First the runtime of the algorithm increases almost linearly (to that of N) as the reference set size increases. Second, there appears to be a knee in the performance curve for small reference set size. Runtime increases linearly to N because a larger reference set introduces more comparisons while building the vectors. Moreover, the knee in the curves signifies an optimum reference set size, around 1, which is explained by the properties of data set A. Data set A is synthetically generated and observation of the distances between pairs appears to be uniformly distributed. The data appear as if they are in a single large cluster, thus a small sample size is enough as predicted by lemma 3. For larger reference set sizes, the cost of building vectors is larger especially for this algorithm, since the expensive $treedist(.)$ function is invoked for this purpose. Moreover, as the distance threshold increases, large documents are not pruned away by the application of l_t; this causes large documents to be verified using the expensive $treedist(.)$ function and this dominates the overall computation.

Figure 9(b) presents the results for RSB. The observations are similar; the additional filtering applied in this algorithm, however, achieves more effective pruning. Even in the case of a large distance threshold, for the range of reference set sizes evaluated in the figure, the algorithm is much faster than N (and subsequently RS). Finally, Figure 9(c) presents the results for RSC. Improved filtering and less expensive computations during the construction of vectors, gives a significant performance advantage to this algorithm. It is much faster than competitors and manages to outperform N even for large distance thresholds and very large reference set sizes.

In Figures 9(d)(e)(f) we present the results of the same experiment, using data set B. Data set B is artificially constructed to contain 8 distinct clusters. The overall performance trends as well as the relative performance of the algorithms remains the same. However, in this case, we can clearly see that the optimal reference set size is around 8, as predicted by Lemma 3. For algorithm RSB (and to some extent algorithm RSC) and for the same values of the distance threshold, performance appears linear (to that of N) in the size of the reference set. The reason is that the additional filtering step applied with the bounds of theorems 1,3 in these algorithms is very effective in these cases. A large number of pairs is pruned by these filters as opposed to the sole application of l_t, u_t in the case of RS. For this data set the performance benefits of RSC are very large. For all distance thresholds and reference set sizes, runtime is below 0.2 of that of N. In addition, the overhead imposed by larger (or smaller) than optimal reference set size is not significant as the involved evaluations are relatively cheap.

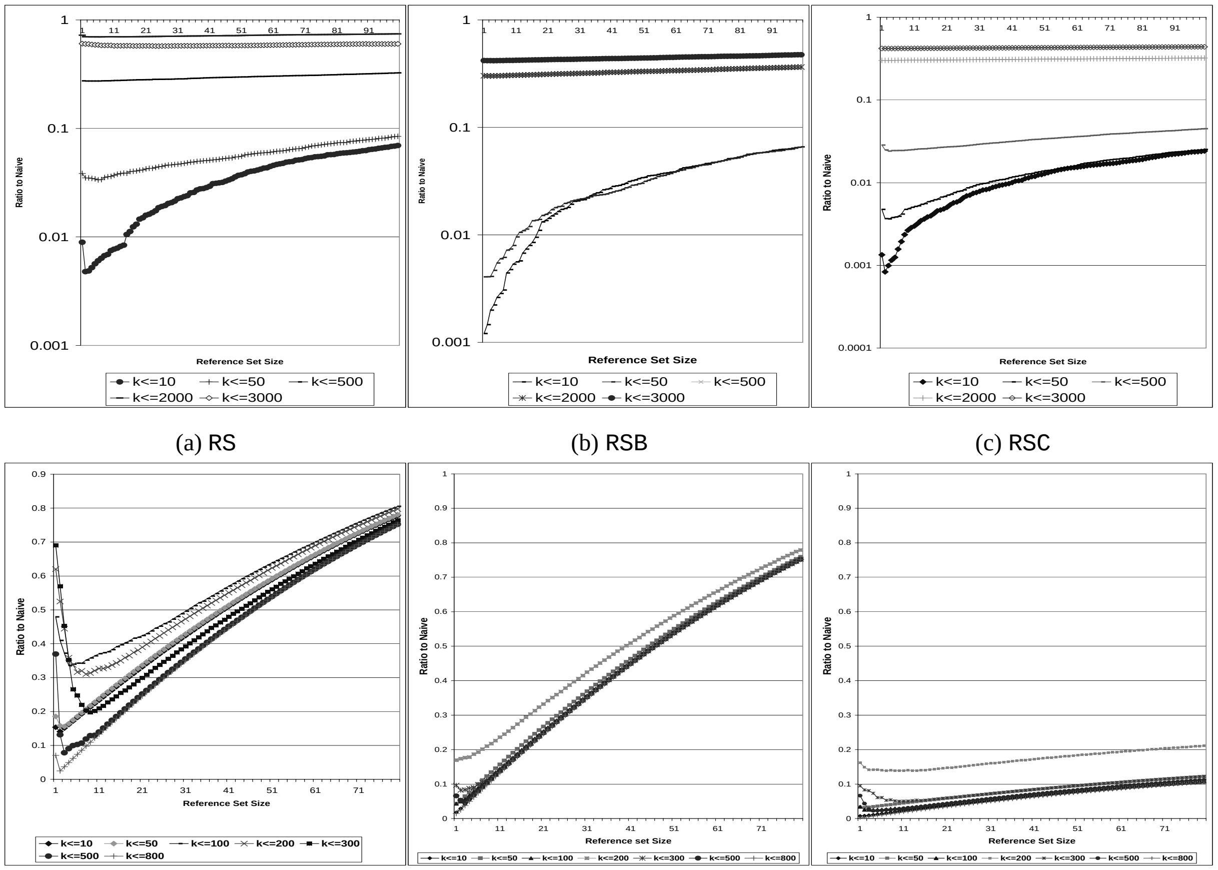
The second experiment we report investigates the performance of the algorithms (B, RS, RSB, RSC) as a function of the distance threshold for various reference set sizes. Figure 10(a)(b) presents the results for data set A for a reference set of size 5 (Figure 10(a)) and 100 (Figure 10(b)). In all cases there is a clear "bell" shaped curve which is explained by the impact of different filters applied as the distance threshold increases. For small distance threshold, the lower bounds applied by the algorithms are effective, in the sense that they can prune away a lot of pairs that don't belong to the answer set. For large distance thresholds the upper bounds are more effective as they can admit a lot of pairs in the answer, saving many tree edit distance computations. Thus the performance curves of all algorithms tail off in small and large distance thresholds, because the filters are extremely effective. In terms of performance, RS appears the worst since its performance is dominated by the expensive $treedist(.)$ invocations while constructing the vectors; in addition pruning solely on the l_t, u_t bounds is not so affective, thus $treedist()$ is invoked on many pairs. Notice that B is more efficient, because it avoids $treedist$ computations by first applying the bounds of theorems 1,3. For the reference set size in Figure10(a), the performance of RSB and RSC are close and clearly outperform all other algorithms. Increasing the reference set size in Figure 10(b) we observe that RSB becomes worse than B because running time is dominated by expensive $treedist$ computations during the construction of the vector sets. RSC appears to be the algorithm of choice in this case as well. Even when the reference set size is much larger than the optimal size (in this case close to 1) the algorithm can still outperform all competitors because the penalty incurred by increased cost during construction of the vectors sets in much smaller.

Figures 10(c)(d) present the results of the same experiment for data set B. Overall trends remain similar, with the "bell" shaped trend explained as before; with eight clusters in data set B, a reference set size less than 8 (Figure 10(c)) makes RSB slightly faster than B, due to less overhead of $treedist$ computations. RSC is still the algorithm of choice even with less than optimal reference set size. As the reference set size increases to 8, all algorithms improve, but RSB starts becoming worse than B for small distance thresholds, because the performance of the algorithm is dominated by expensive $treedist$ computations. For an even larger reference set size, B becomes much better. In all cases, RSC is the algorithm of choice.

7.3 Evaluating Reference Set Selection

To demonstrate the effectiveness of our proposed algorithms and analysis for the choice of the reference set, we conducted the following experiments. For a variety of data sets, we measured the response time as the distance threshold increases, for various sizes of a reference set chosen using a random strategy as well as chosen using the algorithm proposed in Section 5.1 for the case of a known reference set size. In all cases, the proposed algorithm which derives a sample based on Lemma 4 and then clusters to half the threshold distance with a clustering algorithm of choice, always outperformed the strategy that randomly chooses the reference set of a specified size. Figure 11(a) presents representative results for data set B with a reference set size of 8, showing the performance of algorithm RSC, as a fraction of that of algorithm N, as the distance threshold increases. We can observe that RSC is much faster and the effects of filtering are even more pronounced (the curve tails off much more quickly) signifying the choice of a more effective reference set. Similar results were obtained for the other data sets as well.

For the case of an unknown reference set size, we conducted the following experiment. For a variety of real and synthetic data sets, given a join threshold τ we first draw a sample of size at least



(a) RS (b) RSB (c) RSC

(d) RS (e) RSB (f) RSC

Figure 9: Increasing reference set size for datasets A and B

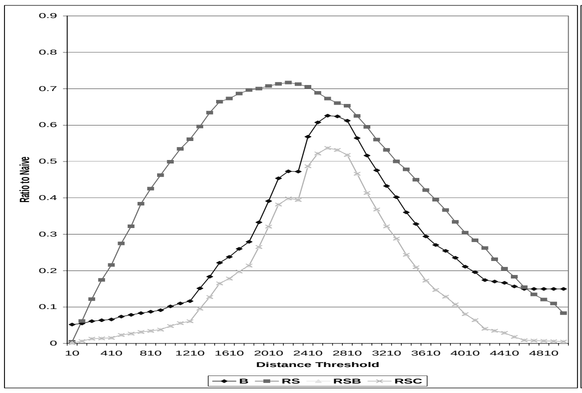
$O(\sqrt{N}\log|N|)$ (for a data set of size $O(N)$) and then iteratively compute clusters, picking the reference set size using Lemma 4. We then compared the reference set size so computed with the known optimal set size (in the case of synthetic data sets) or with the one estimated by observation of performance curves. In all cases, the reference set size we select is very close to the optimal one. Figure 11(b) shows the outcome of this experiment for data set B. The data set contains 8 well separated clusters and we report the number of reference set points our algorithm computes as a function of the distance threshold τ. For a range of τ values the reference set size selected is very close to the optimal. As τ increases, the size computed by the algorithm becomes smaller. This is explained by observing the distribution of document distances in the data set. The minimum distance between clusters is around 250 in this data set. Thus, twice above this threshold, the clusters computed in the sample compared with the ones in the original data set start to mix and as a result the reference set size reported drops. However, contrasting this range of τ values and reference set size with the performance curves reported in Figure 9(f), we observe that in this range a suboptimal reference set size has little impact in performance.

7.4 Performance Results on Real Data

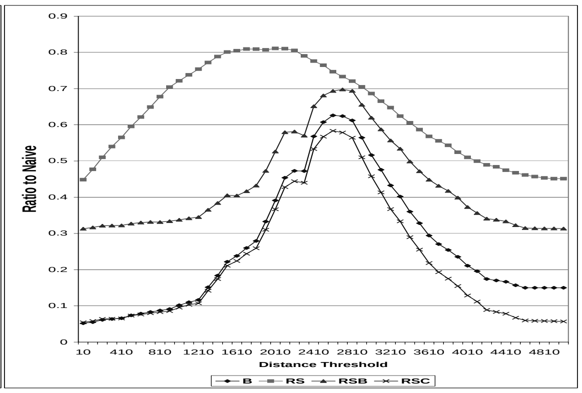
The experiments presented established the performance advantages of algorithm RSC for a wide range of parameters affecting the performance of the algorithms. We used synthetic data sets in order to have the benefit of flexibility in setting these parameters. In this section, we present results on the performance of this algorithm using real data sets of large size. In particular we present performance results for a self join operation, for varying thresholds for the DBLP data set. Running N on this data set is terribly inefficient. Our calculations indicate that it would take more than 81 days on a high end machine to perform a self join on the entire data set. Thus, we first present comparative results for a subset of DBLP records in Figure 12(a). To keep the subset small, but each XML document sizeable, we construct the subset by grouping together all the XML documents corresponding to publications in the same conference, into a single document. We select a subset consisting of 100 conferences. The resulting XML data set is of size 2.2MB. In this case, N requires approximately 3.5 hours to complete. The “bell” shaped curve is evident and is explained due to the effect of filtering as before. Clearly the performance benefits of the proposed algorithm are very large. Figure 12(b) presents the performance of RSC for the entire DBLP data set as a function of the distance threshold. The size of the reference set for each value of the distance threshold, is computed using our proposal in Section 5.1. Response time increases smoothly with increasing distance threshold.

8. CONCLUSIONS

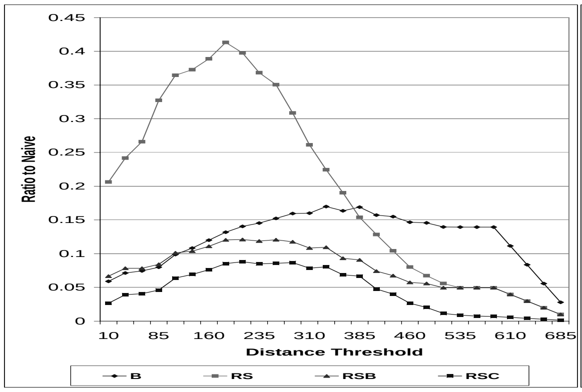
The projected prevalence of XML will inevitably impact several data integration applications. To this end, in this paper, we considered metrics for quantifying distance between XML documents



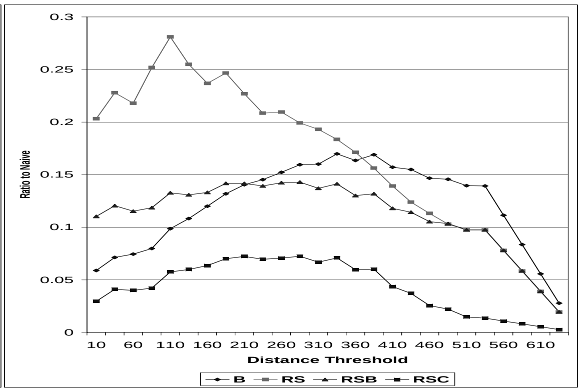
(a) Dataset A reference set size 5



(b) Dataset A reference set size 100



(c) Dataset B reference set size 4



(d) Dataset B reference set size 8

Figure 10: Increasing distance threshold

and subsequently proposed algorithms to perform join operations between XML data sources based on these metrics.

Our work makes the following specific contributions: we proposed lower and upper bounds for tree edit distance between ordered labeled trees, that are computationally more efficient. We have presented a generic technique based on the notion of reference sets that can be used to process joins between XML data sources. We presented this approach along with an analysis on the selection of the reference set. A particular appealing feature of our proposal is that it can incorporate *any* proposal quantifying differences between trees as long as it is a metric. Although we choose to use our technique in conjunction with tree edit distance because of its generality and wide acceptance, any other metric that is more meaningful in a specific application context can be applied. Combining our bounds with properties of the resulting metric space, we proposed various algorithms for processing approximate XML joins and we experimentally quantified the performance tradeoffs.

Several issues for further exploration and experimentation are raised by this work. First, due to the generality of our distance join framework, it would be worthwhile to incorporate in our framework and experiment with other distance metrics as well, capable of quantifying distance between XML documents. It would be interesting to understand the impacts both in performance as well as quality of the results observed in specific application scenarios. Second, indicies have been traditionally applied in databases to speed up the performance of various database operations. Towards this direction it would be worthwhile to explore the application of various indexing schemes proposed for general metric spaces [14][4] to the approximate XML join problem. We plan to investigate these directions in our future work in this area.

9. ACKNOWLEDGEMENTS

We wish to thank the anonymous referees for their comments.

10. REFERENCES

[1] A. Apostolico and Z. Galil. *Pattern Matching Algorithms*. Oxford University Press, 1992.

[2] S. Chawathe and H. Molina. Meaningful Change Detection in Structured Data. *Proceedings of ACM SIGMOD*, May 1997.

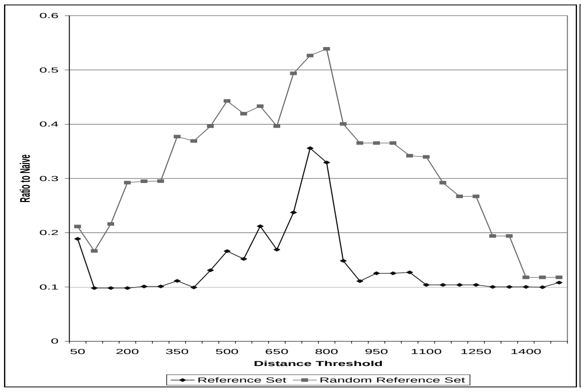
[3] S. Chawathe, A. Rajaraman, H. Molina, and J. Widom. Change Detection in Hierarchical Structured Information. *Proceedings of ACM SIGMOD*, May 1996.

[4] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An Efficient Access Method for Similarity Search Metric Spaces. *Proceedings of VLDB*, pages 426–435, Aug. 1997.

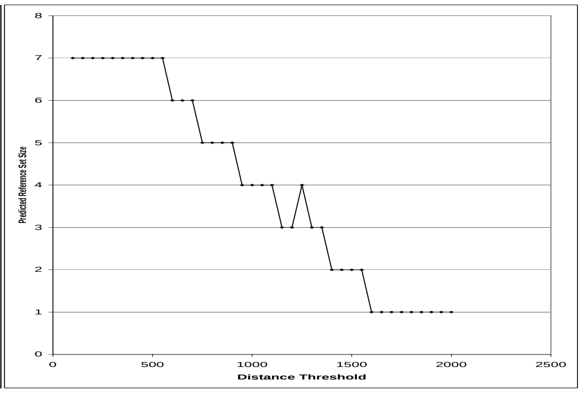
[5] G. Cobena, S. Abideboul, and A. Marian. Detecting Changes in XML Documents. *Proceedings of ICDE*, 2002.

[6] H. Garhaldas, D. Florescu, D. Shasha, E. Simon, and E. Saita. Declerative Data Cleaning. *Proceedings of VLDB*, 2001.

[7] L. Gravano, P. Ipeirotis, H. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava. Approximate strings

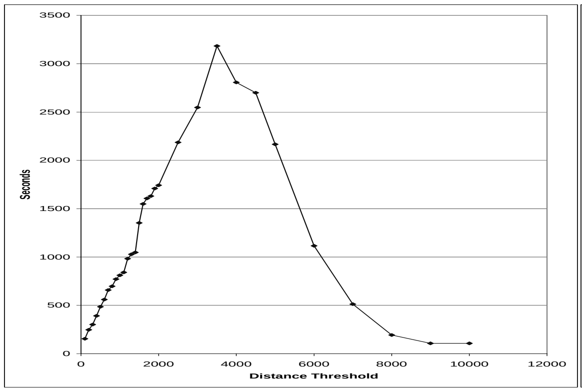


(a) Fixed Reference Set Size

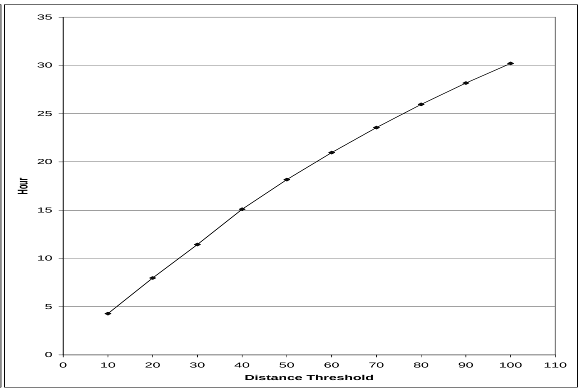


(b) Unknown Reference Set size

Figure 11: Evaluating Reference Set Selection



(a) Increasing Distance threshold on a sample



(b) Increasing Distance threshold on `DBLP`

Figure 12: Performance curves for `DBLP` collection

joins in a database (almost) for free. *Proceedings of VLDB*, 2001.

[8] S. Guha, R. Rastogi, and K. Shim. CURE: An Efficient Clustering Algorithm for Large Databases. *Proceedings of ACM SIGMOD*, pages 73–84, June 1998.

[9] V. Levenshtein. Binary Codes Capable of Correcting Insertions, Deletions and Reversals. *Cybernetics and Control Theory*, pages 707–710, 1966.

[10] A. Marian, S. Abideboul, G. Cobena, and L. Mignet. Change Centric Management of Versions in an XML Warehouse. *Proceedings of VLDB, Rome Italy*, 2001.

[11] G. Navarro. A Guided Tour to Approximate Strings Matching. *ACM Computing Surveys*, Mar. 2001.

[12] D. Sankoff and J. Kruskal. *Time Warps, String Edits and Macromolecules: The Theory and Practice of Sequence Comparison*. Addison-Wesley, Reading, Mass.,, 1983.

[13] S. Sarawagi. Special issue on data cleaning. *IEEE Data Engineeing Bulleting, 23(4)*, 2000.

[14] P. Yianilos. Data Structures and Algorithms for Nearest Neighbor Search in General Metric Spaces. *Proceedings of the Third Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 311–321, Oct. 1992.

[15] K. Zhang. A New Editing Based Distance Between Unordered Labelled Trees. *4th Annual Symposium, Combinatorial Pattern Matching*, 1993.

[16] K. Zhang and D. Shasha. Tree Pattern Matching. *Pattern Matching Algorithms, Apostolico and Galil Editors, Oxford Univesity Press*, 1997.

[17] T. Zhang, R. Ramakrishnan, and M. Livny. BIRCH: An Efficient Data Clustering Method for Very Large Databases. *Proceedings of ACM SIGMOD, Montreal Canada*, pages 103–114, June 1996.